DEFENSE INFORMATION INFRASTRUCTURE (DII)

COMMON OPERATING ENVIRONMENT (COE)

Supplemental Consolidated DCE Application Development Tools Programmer's Guide
Version 1.0.0.0

December 20, 1996

Prepared by:
LOGICON, Inc.
1831 Wiehle Avenue, Suite 300
Reston, Virginia 22090

Distributed Computing Environment

Supplemental Consolidated DCE 1.1 Application Development Tools Programmer's Guide Version 1.0.0.0

December 20, 1996

Defense Information Systems Agency
Joint Interoperability and Engineering Organization
Center for Standards
Information Processing Standards Department

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The Defense Information Infrastructure (DII) Common Operating Environment (COE) Supplimental Consolidated DCE 1.1 Application Development Tools Programmers Guide provides instructions on using a set of application programming interfaces (API's) developed to simplify the development of client-server applications that take full advantage of the services of DCE. These API's are collectively called the DCECOE library.

1.1 Background

The objective DII environment is a tiered, open, and distributed software architecture built upon a client/server model, which allows the separation of data, communications, and display software. To accomplish this, DII has defined a COE that includes support applications, platform services, and reusable software components. To assist in the development of mission applications, the COE provides integrated services to support the mission application software requirements and software development environment.

The DII COE includes distributed computing services to provide specialized support for applications that may be dispersed among computer systems in the network but must maintain a cooperative processing environment. The commercial software selected by the DII COE to provide these services is the Open Software Foundation's (OSF[1]) Distributed Computing Environment.

1.2 Scope

This document is a practical guide to programming applications for the DII COE DCE. It is intended for application developers who have basic knowledge of the concepts and services of DCE, but do not have a detailed understanding of the API's provided by the basic DCE product. This documentation will augment, not replace, the OSF and TRANSARC documentation. The following documents from the reference list in Section 1.5 are suggested companion documents to this guide:

- *OSF DCE Application Development Guide - Introduction and Style Guide*
- *OSF DCE Application Development Guide - Core Components*
- *Guide to Writing DCE Applications, Second Edition*

1.3 Applicability

The information in this document relates to OSF DCE 1.1 and related updates included in DII COE Version 3.0.

[1] The acronym OSF also stands for the DII Operational Support Facility. Unless specifically qualified, the acronym as used in this guide will refer to the Open Software Foundation.

1.4 Report Organization

This guide contains the following sections:

- Introduction - Provides a background of the DII COE Suppimental Consolidated DCE 1.1 Application Development Tools Programmers Guide, its scope, and report organization.
- Overview - Provides an introduction to the rationale and basic concepts of the DCE API's for COE (DCECOE).
- Application Development - Provides an overview of the steps required to build a DCE application using the DCECOE library.
- Appendix A - DCECOE man pages.
- Appendix B - Sample Application.

1.5 References

- *OSF DCE Application Development Guide - Introduction and Style Guide, Open Software Foundation.*
- *OSF DCE Application Development Guide - Core Components, Open Software Foundation.*
- *OSF DCE Application Development Reference - Volume 1, Open Software Foundation.*
- *Guide to Writing DCE Applications, Second Edition, O'Reilly & Associates, Inc.*
- *DII COE Integration and Runtime Specification (I&RTS), Version 2.0, October 23, 1995.*
- *DII COE Integration and Runtime Specification, Appendix X, Distributed Computing Environment, February 6, 1996.*

2. OVERVIEW OF THE DCECOE LIBRARY

This section is intended as an overview of the DCECOE components. It serves as a reference for the function a component performs and why the component is necessary. It is important to remember that the DII COE uses DCE version 1.1.

2.1 Purpose

The DCECOE library is designed to facilitate the development and deployment of manageable, robust DCE client-server applications.

2.2 Basic Concepts

In order to make the DCECOE-based applications more manageable, configurable and to guarantee non-interference, the DCECOE relies heavily on the concept of an application repository which contains many of the meta operations, configuration options, structure, and dependencies of these applications. This section describes some of the concerns that a typical DCE developer will encounter, and then shows how use of the DCECOE library simplifies the development.

2.2.1 Concerns in Building DCE Applications

Building a robust DCE application usually requires many similar steps and requires the use of many of the numerous API's provided in the standard DCE product. The developer will have many choices to make in the design of their application. Developers will each make their own choices based on factors such as experience, style, and application requirements. The following are the basic steps required, and some of the typical concerns.

2.2.1.1 Define the interfaces

The first step in developing a DCE application is to define the client-server interfaces using the DCE Interface Design Language (IDL). Some of the concerns include:

- How many interfaces are needed?
- What are their parameters?
- Does it require any special IDL support
- What RPC semantics should be used?

The result may be one or more IDL files

2.2.1.2 Determine client-server access/naming model

The next step is to determine the method for clients to locate and access servers using the facilities of DCE. Concerns include:

- Which DCE service model will be used?

- Which DCE resource model will be used?
- The number of servers required?
- How will client locate the server?
- Can all servers be registered in CDS?
- How many servers does client need to contact?
  - One? Any one? A specific one? Multiple?
- Which kinds of naming elements in CDS are used?
  - entry, group, profile
- What names are used to identify servers?

(reference to DCE reference materials on design decisions)

2.2.1.3 Determine security requirements

If the application has security or privacy requirements, then the security features of DCE should be utilized. Concerns include:

- How are server principals assigned and used?
  - If the server is run by a user.
  - If the server is automatically run through a noninteractive login.
- How many servers are available?
- How do servers trust each other?
- How does client know the name of the server?
  - Multiple principals
- What security settings does server demand?
- How does the client negotiate appropriate security settings with the server?

2.2.1.4 Develop server

A typical server implements logic to satisfy the interface defined in the first step. However, in addition, the server must implement a variety of other routines, including the following:

- Server initialization processing
  - Registration with runtime
  - Registration with endpoint mapper
  - Registration in CDS
  - Support for serviceability used to manage error messages
  - Support for remote serviceability
- Reference monitor to make access control decisions
- Management authorization routines to start/stop the server
- Auditing routines to manage the capture of audit messages.
- ACL manager to maintain access control information.
  - Database preparation
  - ACL initialization
  - ACL name to object resolution

2.2.1.5 Develop client

The client application makes use of the service interfaces offered by the server, but in addition it must implement additional logic to initialize and manage the DCE interfaces. These include:

- Binding preparation to locate and bind to a server
- Logic to create and maintain authentication information
- Routines to prepare and maintain a serviceability interface for messages.

2.2.2 How DCECOE Simplifies These Concerns

The I&RTS DCE Appendix X defines many of these choices in order to promote uniformity, consistency and to avoid conflicts that can occur when a large number of applications attempts to share a set of common resources (CDS, Security, file systems etc.)

The DCECOE attempts to provide additional support for a wide range of client server development tasks and attempts to provide extensibility using the server configuration record and through the use of optional callbacks.

The following example is representative of performing server initialization. This short code fragment handles all of the details of server registration, login, ACL initialization, auditing, etc.

The include file, **dcecoepublic.h** is located in the **/usr/include/dcecoe** directory and contains all of the information necessary to use the DCECOE interfaces.

```
#include <dcecoe.h>
#define SEGMENT "CALC"
main(int argc, char **argv)
{
    error_status_t st;
    COEDCEinitialize_server(SEGMENT,
           S_LOGIN|S_REFRESH|S_KEYMGMT|S_ACL|S_AUDIT|
           S_CDSEXPORT|S_LISTEN|S_CLEANUP|S_MGMTAUTH,
           NULL, &st);
    exit (st != rpc_s_ok);
}
```

This initialization sequence performs up to several thousands of lines of complex DCE logic which your application need not contain. The actual functions performed are selected by using the 'flags' parameter. The manual pages contain the detailed information about each of the options.

You are probably wondering how this application which has a single constant -- the SEGMENT name -- knows about the dozens of constants and parameters that are required to satisfy the DCE APIs. This information is recorded in the **dced**'s configuration record under the record name **CALCserver**. This information gets installed in the **dced** when the server application is installed on a particular machine.

2.3 DCECOE Procedures

Functions in the DCECOE RPC library provide a simplified mechanism for developing client-server applications using the OSF DCE services. All of the procedure calls required to initialize a DCE client or server are consolidated into a single procedure call. The routines make use of sensible, but overridable, defaults to simplify the development of applications. The DCECOE library takes advantage of and provides easy access to many of the features of OSF DCE Version 1.1, as described below.

The DCECOE library consists of the routines listed below and described in separate man pages.[2]

2.3.1 Server Procedures

COEDCEinitialize_server() - Initializes a DCE server
COEDCEsignal_server() - Signal a server to enter listen loop
COEDCEcreate_acl() - Creates an access control list (ACL)
COEDCEis_auth() - Makes an authorization decision
COEDCEfinalize_server() - Terminate server resources

2.3.2 Client Procedures

COEDCEinitialize_client() - Initializes a DCE client
COEDCElocate_server() - Locates a server
COEDCEgetvector() - Retrieves a binding vector
COEDCEinquire_server() - Gets info about a server
COEDCEstart_server() - Prepare a handle for communications with a server
COEDCEfree_servers() - Frees a server
COEDCEfinalize_client() - Frees allocated resources

2.3.3 Features

2.3.3.1 Security

The security of an RPC connection can be configured to any level from unauthenticated to authenticated and encrypted. The DCECOE routines automatically perform server login, authentication, password maintenance, and security context refresh. They guarantee that clients use appropriate security choices as required by servers. DCE security mechanisms are used to identify and authenticate servers rather than inquiring of servers themselves for security identification.

[2] The prefix for these routines will be changed to DCECOE in the next version, to be consistent with the overall library name. The current names will be retained for the next two releases.

2.3.3.2 ACL Management/Reference Monitor

The server routines include an access control list (ACL) manager to allow remote management of ACL's for the server. An application may define its own functions to be controlled (e.g. read/write/delete for storage, print/control for a printer, view/update for document, etc.). At built-in reference monitor makes access decisions based on the contents of the ACL database and performs access auditing.

2.3.3.3 Server Administration

The DCECOE server library implements a management interface that allows the server to be remotely managed using the standard dcecp. Server configuration information, including security parameters, file locations, and application configurables, are maintained as extended attributes within CDS.

2.3.3.4 Server Startup

The DCECOE client library provides functions to allow a server to be started on demand if one is not currently running.

2.3.3.5 Server Connection and Query

The client library allows the client to connect to any available server, or to locate all available servers and retrieve information about the servers in order to make a connection decision. Decisions can be made based on the availability of the server, the 'objects' maintained by the server, or any other information agreed upon between the client and server and recorded in the configuration information within CDS.

2.3.3.6 Auditing

The server routines maintain an audit file that can be written by the reference monitor or the application using standard OSF DCE audit functions.

2.3.3.7 Serviceability Messages

The DCECOE library functions make use of the OSF DCE 1.1 serviceability interfaces to generate and manage error messages. The server management interface allows messages of different severity to be turned on or off and routed to different locations (e.g. error log, stderr, etc.).

3. BUILDING A DCE APPLICATION

This section describes the steps required to build a DCE application using the DCECOE library. The discussion assumes that the reader is familiar with the background information from Chapters 1 and 2 of the reference *Guide to Writing DCE Applications*. The discussion is based on the sample calculator application supplied with the DCECOE library. This application is intended to be used as a template for developing DCE applications.

3.1 Defining Interfaces (.idl/.acf files)

3.1.1 Get a UUID

Each DCE interface has a unique identifier (uuid) to ensure compatibility of the client and server. DCE will only allow a binding between compatible interfaces. Get a unique identifier for each interface to be defined, using **uuidgen**. The uuid information will become part of the IDL file that defines the interface. An example uuid for the sample calculator application is as follows.

```
[uuid(0073a028-fbdb-1e53-908e-08002b13ca26), version(1.0)]
```

3.1.2 Write the IDL file

Define the interfaces in a IDL (.idl) file. Insert the uuid from the previous step into the .idl file. The interface definition is the same for a DCECOE application as for any other DCE application. NOTE: If you are using the sample application as a template, you will need to replace the uuid in the calc.idl file with a new uuid for the new interface.

The following is an extract from an IDL file defining the calculator interface. The full sample is included in Appendix B. The IDL file defines the calculator operations (e.g. add and subtract) as well as the parameters for each operation. The IDL file also defines constants that will be used in the client and server applications, such as return status codes (e.g., calc_s_ok).

```
interface calculator
{
 const long calc_s_ok        = 0;
 const long calc_div_by_zero = 100;
 long add (
   [in]    long a,
   [in]    long b,
   [out]   error_status_t *st
 );
 long subtract (
   [in]    long a,
   [in]    long b,
   [out]   error_status_t *st
 );
```

```
}
```

For more information about defining IDL files, please refer to *Guide to Writing DCE Applications* Chapter 2 or *OSF DCE Application Development Guide - Core Components* Chapters 17 and 18.

3.1.3 Define Access Control File

The access control file (.acf) is usually optional, but is required when using the DCECOE library. In the sample application, the ACF file specifies that the binding handle is managed by the client application and explicitly passed as part of the interface. DCE will automatically include the binding handle in the argument list, even though it is not included in the interface definition in the IDL file. This option is necessary in order to use security features. The sample ACF file also informs DCE to report any communications errors in the defined status parameter **st**. The ACF file for the sample application is shown below.

```
/* Sample Application */

[ explicit_handle ]
interface calculator
{
   add([comm_status] st);
   subtract([comm_status] st);
}
```

The *OSF DCE Application Development Guide - Core Components* Chapter 18 provides additional information on using ACF files.

3.1.4 Compile the Interface

Compile the .idl and .acf files using the **idl** compiler. Usually this will be automated by a line in the application make file. This creates a header file for the interface (e.g., **calc.**h) as well as client and server "stub" files (e.g. **calc_cstub.c** and **calc_sstub.c**).

```
idl -cc_cmd "cc -c" -I/usr/include/dce -I/usr/include/dcecoe \
-I. calc.idl
```

3.2 Defining Serviceability Messages (.sams files)

3.2.1 Define a Component Name

The serviceability messages file defines message text and audit message numbers for use by the application. All serviceability messages are identified by a six-letter sequence identifying the

"technology" and "component" that generated the message.[3] Determine a three-letter component name for the application based on the segment prefix (e.g., **cal** for the sample application). These three letters will appear on every system-generated message from the application. Insert the component name in the front of the SAMS file, as shown in the sample below. There are no differences in defining a SAMS file for a DCECOE application compared to any other DCE application. NOTE: If using the sample application **calc.sams** file as a template, there are numerous places where the component name is used in variable names by convention, and must be changed for a different application.

```
# Part I
# This part defines the lowest-level table, the one that contains
all the
#  messages (defined in the third part) in a straight array.
component       cal
table           cal__table
technology      dce
```

3.2.2 Complete the SAMS File

Develop the serviceability message (.sams) file containing the audit events and messages for the application. The following is an extract from the sample application file **calc.**sams. The entire file is included in Appendix B.

```
# Part II
# This part defines the sub-component table
serviceability  table cal_svc_table handle cal_svc_handle
start
    sub-component cal_s_manager "manager"  cal_i_svc_manager
    sub-component cal_s_server  "server"   cal_i_svc_server
end
#
# Part IIa
# This part contains event codes for auditing
#
start
code  add_event
text  "add operation"
action ""
explanation ""
end
# Part III
# This part defines the serviceability messages.
#
```

[3] Applications are supposed to be identified with the technology **dce** and an identifying number assigned by the OSF. Until a block of numbers are assigned for COE applications, a unique component name derived from the segment prefix should be used.

```
start
code            cal_sad_ending
sub-component   cal_s_server
attributes      "svc_c_sev_error"
text            "server initialize failed"
action    ""
explanation  ""
end
```

For more information on defining SAMS files, please refer to the *OSF DCE Application Development Guide - Core Components,* Chapter 3, and the **sams** man page.

3.2.3 Compile the SAMS file

Compile the .sams file using the **sams** command. This will also usually be part of a make file. The sams program creates as many as 10 files, depending on the options given. In the case of the sample application, only the message header (dcecalmsg.h) and message (dcecal.cat) files are required. The header file is used by any client or server routines that print serviceability messages. The message file is used at execution time and must be delivered with the application segment. The following is an example from the sample application make file.

```
sams -oh calc.sams
```

3.3 Developing the Server

Write the server, making use of the DCECOE library routines. Usually the server setup code should be in a separate file from the "manager" code that implements the application logic of the server. Additional background on developing servers can be found in *Guide to Writing DCE Applications* Chapter 1. The next paragraphs walk through the initialization program and the manager program for the sample application. The complete example is included in Appendix B.

3.3.1 Write the Server Initialization

This program initializes the server and begins listening for clients. The program starts by including the DCECOE public header file and the header file for the server interface. For convenience it also defines the segment prefix for use in the DCECOE API calls.

```
/* Sample server initialization code */

#include <dcecoe/dcecoe.h> /* for use with COEDCE APIs */
#include "calc.h"

#define SEGMENTSERVICE "CALC"
```

For this simple program, the initialization main program consists of a single DCECOE library call, as shown below. The **COEDCEinitialize_server()** routine reads the server configuration

attributes, and performs initialization based on these attributes and the values of flags provided. In this case, the routine performs a DCE login for the server principal, sets up to perform key management, auditing, and server management, and begins listening for client calls. Control remains within the API until the server is signaled to stop.

```
main(int argc, char **argv)
{
    error_status_t st;

    COEDCEinitialize_server(SEGMENTSERVICE,
           S_LOGIN|S_REFRESH|S_KEYMGMT|S_ACL|S_AUDIT|
           S_CDSEXPORT|S_LISTEN|S_CLEANUP|S_MGMTAUTH,
           NULL, &st);

    exit (st != rpc_s_ok);
}
```

For more complex servers, additional initialization logic would be required prior to the DCECOE call to handle initial parameters, process configuration files, and open files or initialize databases.

3.3.2 Write the Manager

The application-specific logic to implement the interface's operations is included in the "manager" program. The manager also includes the DCECOE public include files and the header file for the interface. In addition, it must include the messages header file (e.g. **dcecalmsg.h**) generated by **sams** in order to use the definitions of serviceability messages and audit events.

```
#include <dcecoe/dcecoe.h>
#include "calc.h"          /* build by IDL */
#include "dcecalmsg.h"     /* built by SAMS - audit codes */
```

The remaining portion of the manager consists of the definition of the operations defined for the server interface. The operations are defined much as they would be if they were local subroutines rather than remote procedures. Note that the binding handle **bh** is explicitly included at the front of the argument list, even though it is not present in the IDL definition of the interface.

```
idl_long_int
add (
   rpc_binding_handle_t bh,
   idl_long_int         a,
   idl_long_int         b,
   unsigned32           *st)
{
        /* implementation of add operation omitted */
}
```

```
   idl_long_int
   subtract (
      rpc_binding_handle_t bh,
      idl_long_int         a,
      idl_long_int         b,
      unsigned32           *st)
   {
         /* implementation of subtract operation omitted */
}
```

The actual implementation of the add operation is shown below. The routine initializes two structures used to identify the client who originated the call to the server, and to identify the required permissions[4]. It then calls **COEDCEis_authorized()** to determine if the client has the required permission. Based on the result of that call, the routine returns the result (e.g. **a+b**), or returns an error.

```
      COEDCEclientid_t   client;
      COEDCEobject_t     object;

      /* this is how we identify the client */
      client.identity = ID_HANDLE;
      client.id.handle = bh;

      /* this represents the object to look up and the required
         permissions */
      memset(&object, 0, sizeof(object));
      object.name = "calculator";
      object.permname = "a";    /* add */

      if (COEDCEis_authorized(&client, add_event, &object, NULL, st)
          == aud_c_esl_cond_success)
         return(a+b);
        /* st has status code */
      return -1;
```

The second argument to the **COEDCEis_authorized()** call identifies an audit event to be initialized. Although not shown in this sample, an audit record is initialized by the library and may be written by the application if desired. Further information on auditing can be found in *OSF DCE Application Development Guide - Core Components*, Chapter 33.

[4] The API is being revised to hide these structures inside the COEDCE library and simplify this call. The revised API will be available in the next version.

3.3.3 Compile the Server

Compile the server, to include the server stub created from the IDL file. Link with the **libdcecoe** library. The following illustrates the compile and load statements extracted from the sample application make file.

```
cc -c -g -DSOLARIS -D__EXTENSIONS__ -I. -I../include \
   -o CALCserver.o CALCserver.c
cc calc_sstub.o CALCserver.o CALCmanager.o /usr/lib/libdcecoe.a \
   -L/usr/lib/dce -ldce -lnsl -lthread -lm -o CALCserver
```

3.4 Writing the Client

3.4.1 Develop the Client

Write the client, making use of the DCECOE library routines. The following paragraphs walk through the sample application, which is included in Appendix B.

The client must also include the DCECOE public header file as well as the interface header. For convenience it also defines the segment prefix.

```
#include <dcecoe/dcecoe.h>
#include "calc.h"

#define SEGMENTSERVICE "CALC"
```

The client main program defines local variables required for the DCECOE library and the user interface logic that runs the calculator application.

```
main(int argc, char **argv)
{
    unsigned32           err;     /* COE error */
    error_status_t       dceerr;  /* DCE error */
    rpc_binding_handle_t  handle; /* binding handle */

    /* interface client logic */
    idl_long_int         a, b, c;
    char                 operand;
    error_status_t       st;
    int                  rc;
```

The client then initializes the DCE environment using **COEDCEinitialize_client()**. This logic primarily sets up internal structures for the client. The **CHECK** routine is implemented within the client to check error codes and print an error message if required. See Appendix B.

```
    err = COEDCEinitialize_client(SEGMENTSERVICE, 0, &dceerr);
    if (CHECK(err, "initialize_client", dceerr))
```

```
        exit(1);
```

The sample application contains two different routines to bind to a server, one called **simple()** and one called **complex()**. The simple one requests a single server and attempts to bind to it, while the complex routine looks at all available servers and selects one based on some criteria. Each is described below. The routine to be used is selected by changing the if statement shown below.

```
#if 1
    handle = simple();
#else
    handle = complex();
#endif
```

The simple case is shown below. In makes a single call to **COEDCElocate_server()** requesting a single binding. The library routine will locate a server using the pointer into CDS provided in the client's configuration attributes. See Chapter 4 for more information. The **COEDCEgetvector()** routine is used to return the binding vector containing the results of the **COEDCElocate_server()** call[5].

```
rpc_binding_handle_t
simple(void)
{
    unsigned32     count = 1;
    unsigned32     err;
    error_status_t dceerr;

    err = COEDCElocate_server(0, 0, 0, &count, &dceerr);
    if (CHECK(err, "locate_server", dceerr) || count < 1)
      return NULL;
    else
      return (COEDCEgetvector())->binding_h[0];
}
```

The complex binding routine make more complete use of the DCECOE library capabilities. It also begins with the definition of variables required by the library calls.

```
rpc_binding_handle_t
complex(void)
{
    unsigned32             count = 100;
    unsigned32             one = 1;
    rpc_binding_handle_t   handle = NULL;
    server_t               *servers;
```

[5] The current implementation of COEDCElocate_server() is not thread-safe because the binding vector is maintained in the DCECOE library. The next implementation of the DCECOE library will correct this problem.

```
        int                     i;
        unsigned32              err;
        error_status_t          dceerr;
```

It also uses **COEDCElocate_server()** to find servers, using the CDS pointer configured for the client. In this case it requests that up to 100 server bindings be returned.

```
        err = COEDCElocate_server(C_NOOBJ, 0, 0, &count, &dceerr);
        if (CHECK(err, "locate_server", dceerr) || count < 1)
          return NULL;
```

If there are errors or no servers can be found, the routine returns a NULL handle. If successful, the routine looks at each server in turn, using **COEDCEinquire_servers()** with the **C_**EXEC option to determine if the server is running. If the server is not running, the loop continues to look at the next server. In this example, the routine selects the first running server, however a more complex client could obtain additional information about the server to use in its selection. The **C_**CONF option could also be used to obtain information about all configured servers, whether running or not. Once a server is selected, the list of applicable servers returned from **COEDCEinquire_servers()** is released using **COEDCEfree_servers()**.

```
        for (i=0; i<count; i++) {
            err = COEDCEinquire_server(C_EXEC, 0, &one, &servers,
                          (COEDCEgetvector())->binding_h[i], &dceerr);
            if (CHECK(err, "inquire_server", dceerr) || count < 1)
               continue;

          /* pick one based on some criteria */
          /* (in this example, just select the first one running) */

          handle = (COEDCEgetvector())->binding_h[i];
          err = COEDCEfree_servers(servers, one, &dceerr);
          CHECK(err, "free_servers", dceerr);

          /* we found one we liked */
          if (handle)
            break;
        }

        return handle;
    }
```

Whether the simple or complex routine is used, the result will be a binding handle for a server, or **NULL** is no server is found. In the latter case the program exits.

```
        if (handle == NULL) {
          printf("server not installed correctly\n");
```

```
        exit(1);
    }
```

If a handle is returned, the client binds to the server using **COEDCEstart_server()**. In this case, the flags indicate that the DCECOE library should attempt to ping the server to ensure that it is answering calls, it should attempt to start the server if it is not running, and it should establish a secure connection using the parameters established in the server's configuration record. If it fails to connect, the client exits. A more sophisticated client could include more graceful error handling, such as seeking an alternate server.

```
    err = COEDCEstart_server(C_PING|C_START|C_SECURE, 0,
                             handle, &dceerr);
    if (CHECK(err, "start_server", dceerr))
        exit(1);
```

Now that the client is bound to a server, the client enters a loop where it interacts with the user asking for an operation and two values, and calling the appropriate server operation. Notice that there are no DCECOE routines required during this part of the program, and the calls to the calculator operations (e.g. **add** and **subtract**) are the same as if the operations were local procedure calls, with the exception of the explicit binding handle. The client remains in this loop until the 'q' operation is entered.

```
/* user interaction */
  while (true) {
    fprintf(stdout, "Operation: (op val1 val2) ");
    fflush(stdout); fflush(stdin);
    rc = fscanf(stdin, "%c %ld %ld", &operand, &a, &b);
    if (operand == 'q') break;
    switch (operand) {
    case '+':
      c = add(handle,a,b,&st);
      break;
    case '-':
      c = subtract(handle,a,b,&st);
      break;
    default:
      fprintf(stderr, "Invalid operand\n"); continue;
    }
    if (st == calc_s_ok)
      fprintf(stdout, "%ld %c %ld = %ld\n", a, operand, b, c);
    else
      CHECK_STATUS(st, "operation failed", CONTINUE);
    (void *)fgetc(stdin);
  }
```

The final task for the client is to terminate the binding and free up internal structures.

```
    COEDCEfinalize_client(0, &st);
}
```

3.4.2 Compile the Client

Client compilation is very similar to server compilation. Compile the client, to include the client stub. Link with the **libdcecoe** library. The following is again an extract from the makefile.

```
cc -c -g -DSOLARIS -D__EXTENSIONS__ -I. -I../include \
   -o CALCclient.o CALCclient.c
cc calc_cstub.o CALCclient.o /usr/lib/libdcecoe.a -L/usr/lib/dce \
   -ldce -lnsl -lthread -lm -o CALCclient
```

3.5 Installing the Application

The initialization scripts provided with the distribution are in preliminary form. They define three levels of installation and removal; application, server, and client. Application level is performed once per cell after the application's SEGMENT installation occurs. The server and client portions are used on machines after SEGMENT installation occurs.

Creating an application using the sample requires hand modification of the **Postinstall** script.

The long term goal is to automatically generate the server installation based on a set of DCE descriptors.

3.5.1 PostInstall Scripts

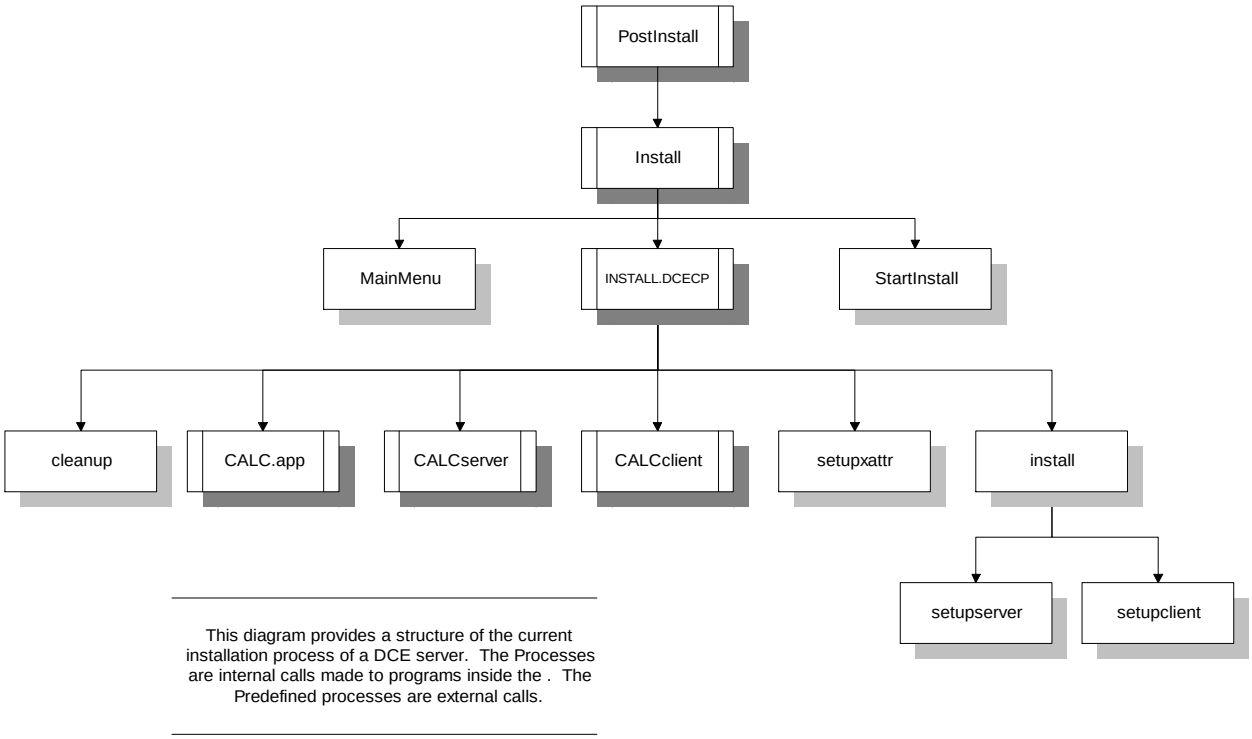

Figure 1 CALC Structure

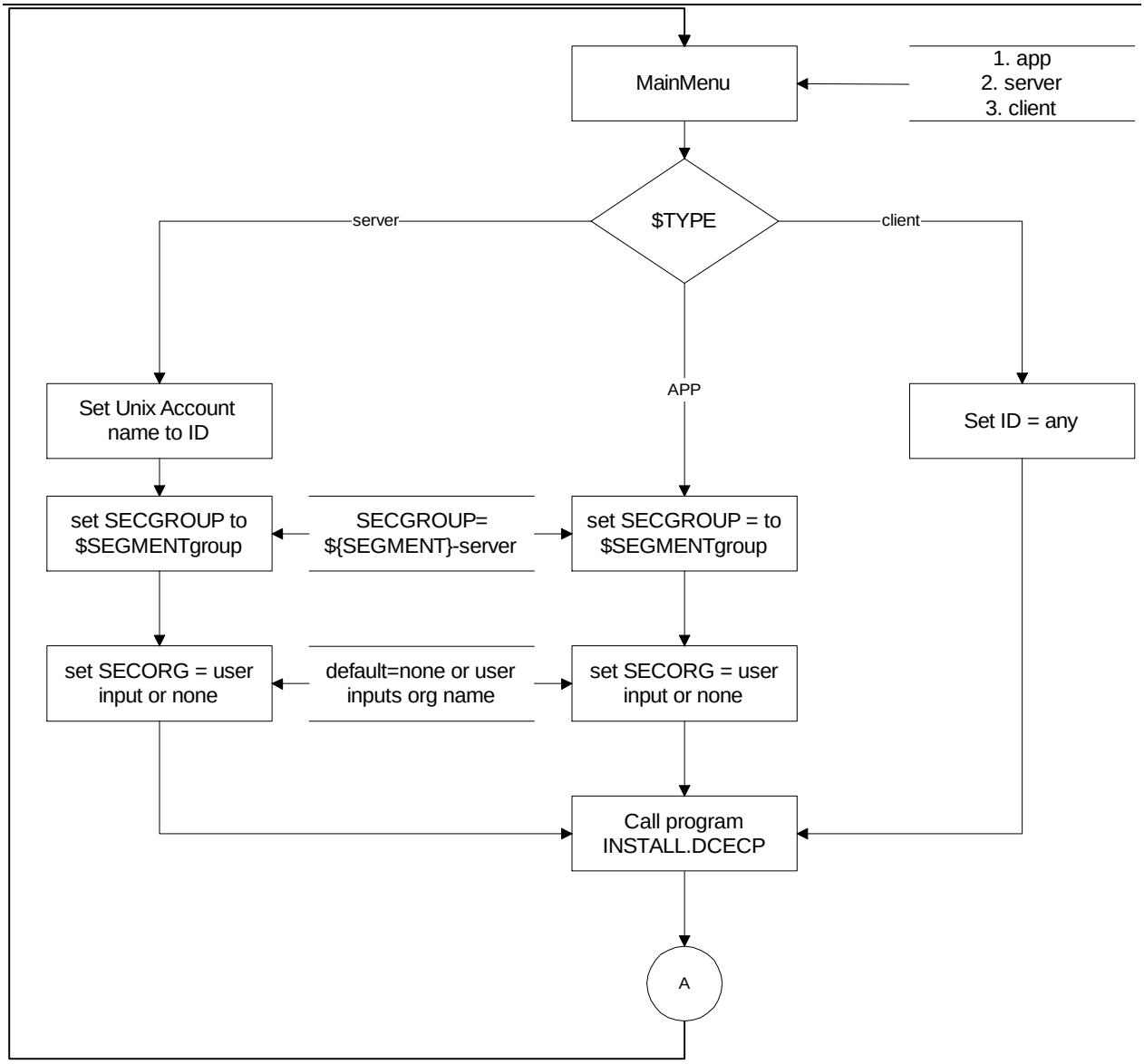

Figure 2 Install

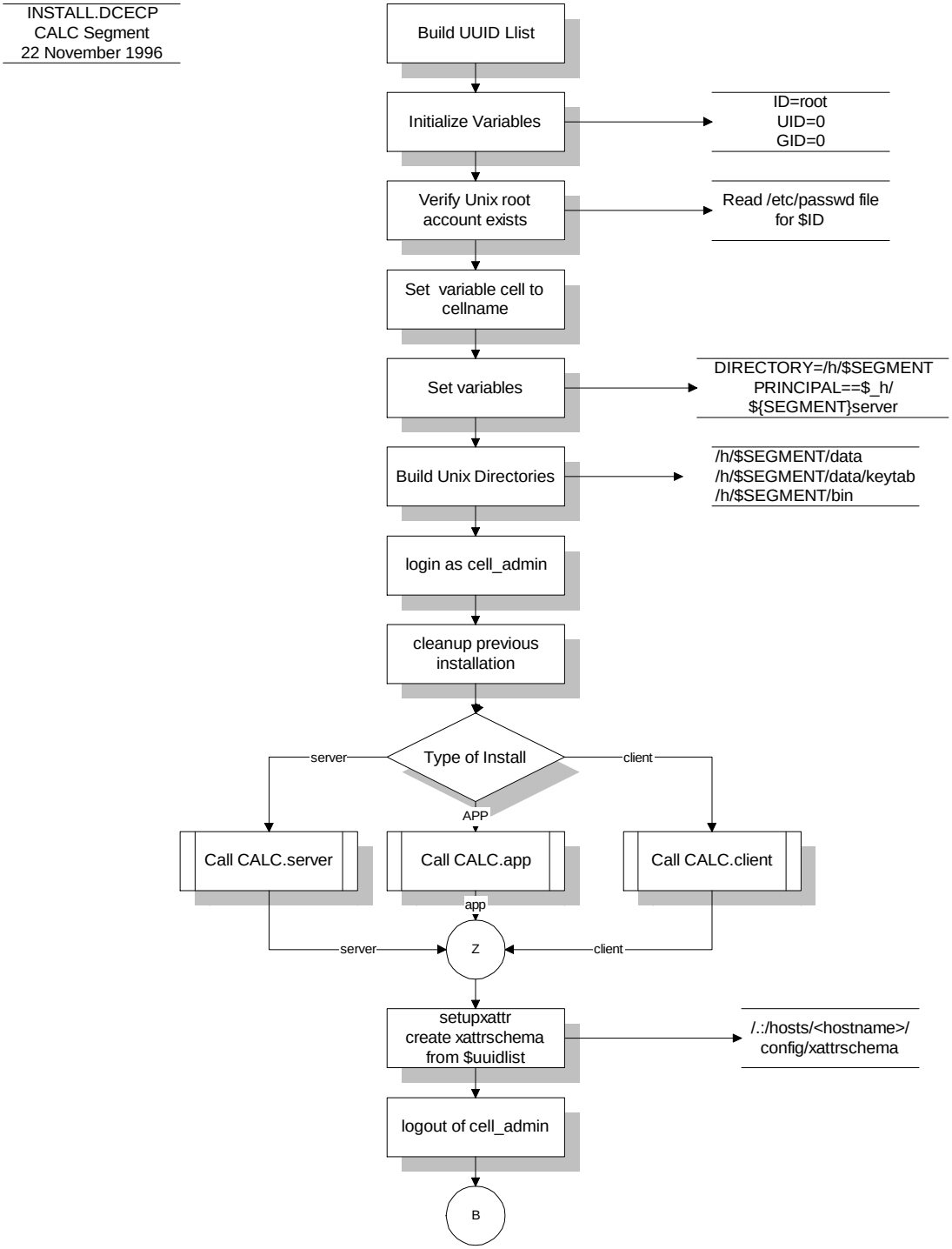
INSTALL.DCECP
CALC Segment
22 November 1996
Build UUID Llist
Initialize Variables
ID=root
UID=0
GID=0
Verify Unix root account exists
Read /etc/passwd file for $ID
Set variable cell to cellname
Set variables
DIRECTORY=/h/$SEGMENT
PRINCIPAL==$_h/${SEGMENT}server
Build Unix Directories
/h/$SEGMENT/data
/h/$SEGMENT/data/keytab
/h/$SEGMENT/bin
login as cell_admin
cleanup previous installation
Type of Install
server
client
APP
Call CALC.server
Call CALC.app
Call CALC.client
app
server
Z
client
setupxattr create xattrschema from $uuidlist
/.:/hosts/<hostname>/config/xattrschema
logout of cell_admin
B

Figure 3 INSTALL.DCECP

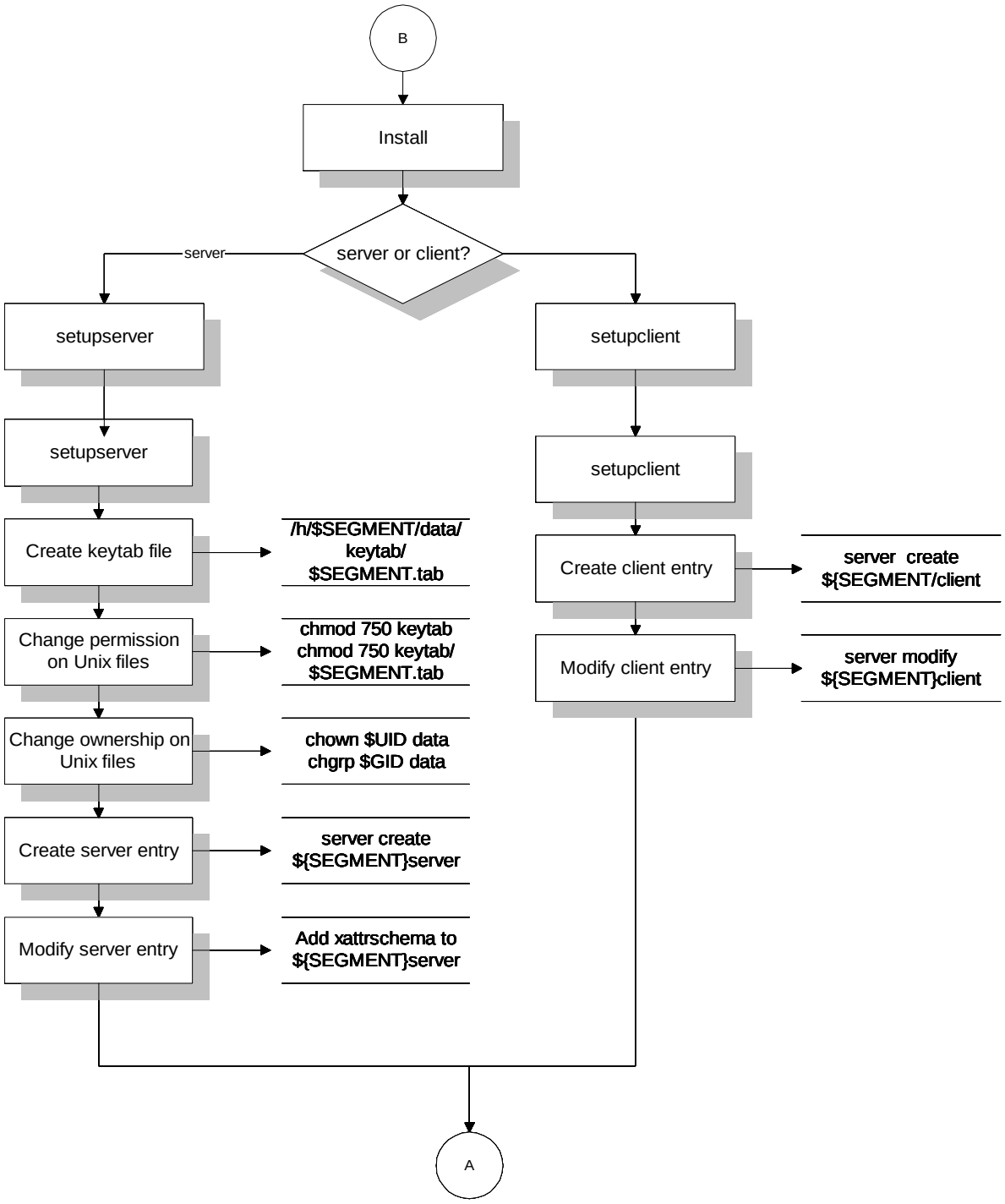

Figure 4 INSTALL1.DCECP

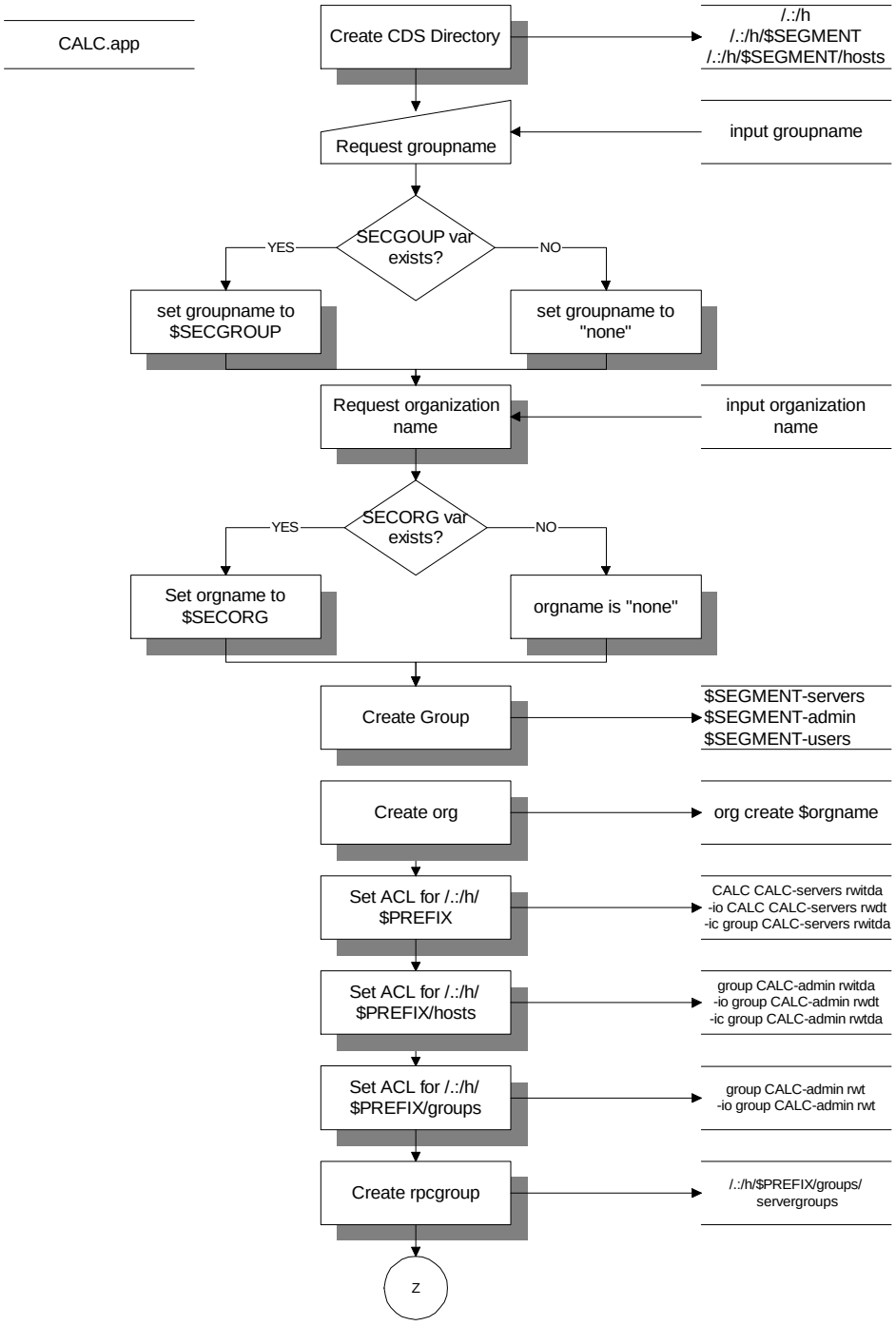

Figure 5 CALC.app

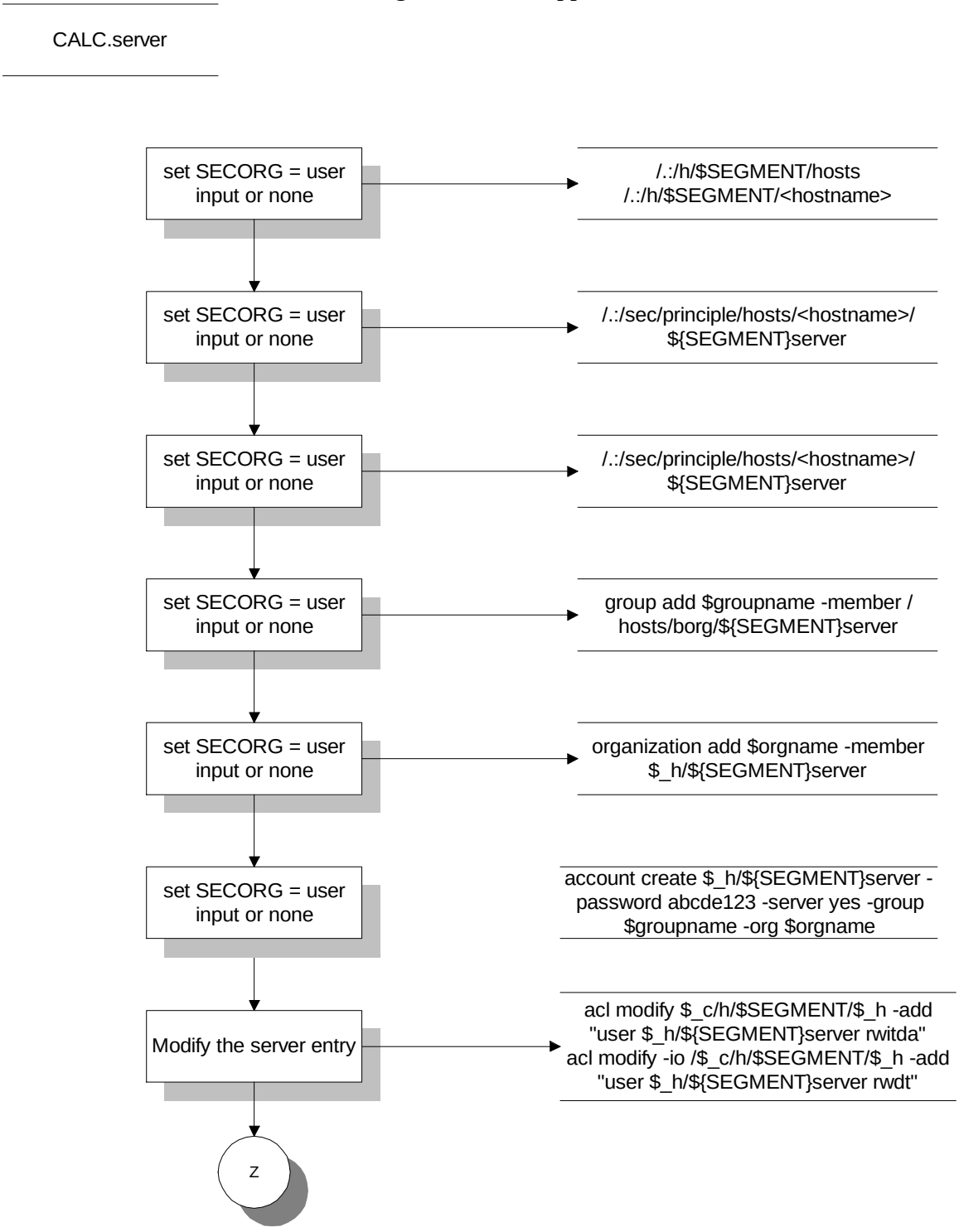

Figure 6 CALC.server

Figure 7 CALC.client

3.5.2 ACL setup Script

TBD

3.6 Additional Examples

The following paragraphs provide additional examples of the use of the DCECOE library under different circumstances.

NOTE: These examples are still under development.

3.6.1 Selection of Server

The complex method of selecting a server previously presented did not exercise the full potential of DCE. The client may use any information available, including that provided by standard DCE calls, in order to select a server. For example, the client could query the values of server attributes in order to determine server capabilities. The client could PING the server and measure the response time as a first-approximation of the "distance" to the server.

The following code fragment shows how the client could determine the hostname of the server if that is needed in selecting a server.

```
rpc_binding_to_string_binding(handle, &sbinding, &dceerr);
/* get the IP address "xx.xx.xx.xx" as a string */
rpc_string_binding_parse(sbinding, NULL, NULL, &ipaddr,
                         NULL, NULL, &dceerr);
rpc_string_free(&sbinding, &dceerr);
/* convert to binary representation */
addr = inet_addr(ipaddr);
rpc_string_free(&ipaddr, &dceerr);
/* retrieve our hostname */
hp = gethostbyaddr((char *)&addr, sizeof(addr), AF_INET);
```

3.6.2 Use of Multiple Servers

The example demonstrates a single client and server as part of the same segment. The DCECOE library makes no assumption about the number of servers used by a client, or the names of the segments.

To use multiple servers, the client **srvrconf** entry must be configured with multiple **services** attributes, one for each service to be used. Each entry must identify the interface to the service, as shown in the example in Section 2.4.6. The second argument of the **COEDCElocate_server()** and **COEDCEinquire_server()** routines is an index of the service in the configuration record.

3.6.3 Three-tier Applications

Applications may be both a client and a server in a three-tier arrangement. The DCECOE library makes no restriction in this regard. NOTE: Since the DCECOE client libraries are not currently thread-safe, the application must take care to serialize the calls to the DCECOE library routines between **COEDCElocate_server()** and **COEDCEgetvector()**.

NOTE: A future version of this guide will provide an example of a three-tier application using multiple servers.

3.6.4 Object-based Binding

The standard DCE has the ability for servers to associate themselves with "objects" (identified by uuid's), and for clients to request a binding to any server providing a specified object. The objects supported by a server are identified within its rpcentry within CDS. This facility is designed to allow the location of coarse-grained objects (e.g. specific branches of a bank, or classes of users). It is not designed for fine-grained objects (e.g. an individual account in a bank).

The DCECOE library allows the use of this capability. The server is responsible for registering supported objects using standard DCE calls. The client must have the uuid's of desired objects

pre-configured within its **services** attribute for the appropriate interface. The third argument to **COEDCElocate_server()** is an index of an object to locate.

NOTE: A later version of this guide will present an example using this capability.

3.6.5 Application-specific Attributes

The idea of extended attributes in the client or server configuration entries is a powerful capability that can also be used directly by the application. Applications may define additional attributes as part of the client or server installation. Values may be assigned to the attributes during initialization, during execution, or by an administrator at any time.

The DCECOE library does not currently provide any convenient facility for obtaining attributes[6], however the application can obtain them using standard DCE API's.

3.7 Structure of DCE Namespace

This section illustrates the use of the DCE namespace by an application using the DCECOE library.

3.7.1 Cell Directory Services (CDS) Namespace

The illustration below shows the structure of CDS after the installation of the sample CALC application segment on host1.

[6] The next version of the DCECOE library will contain functions to allow an application to easily obtain extended attributes about the client or server.

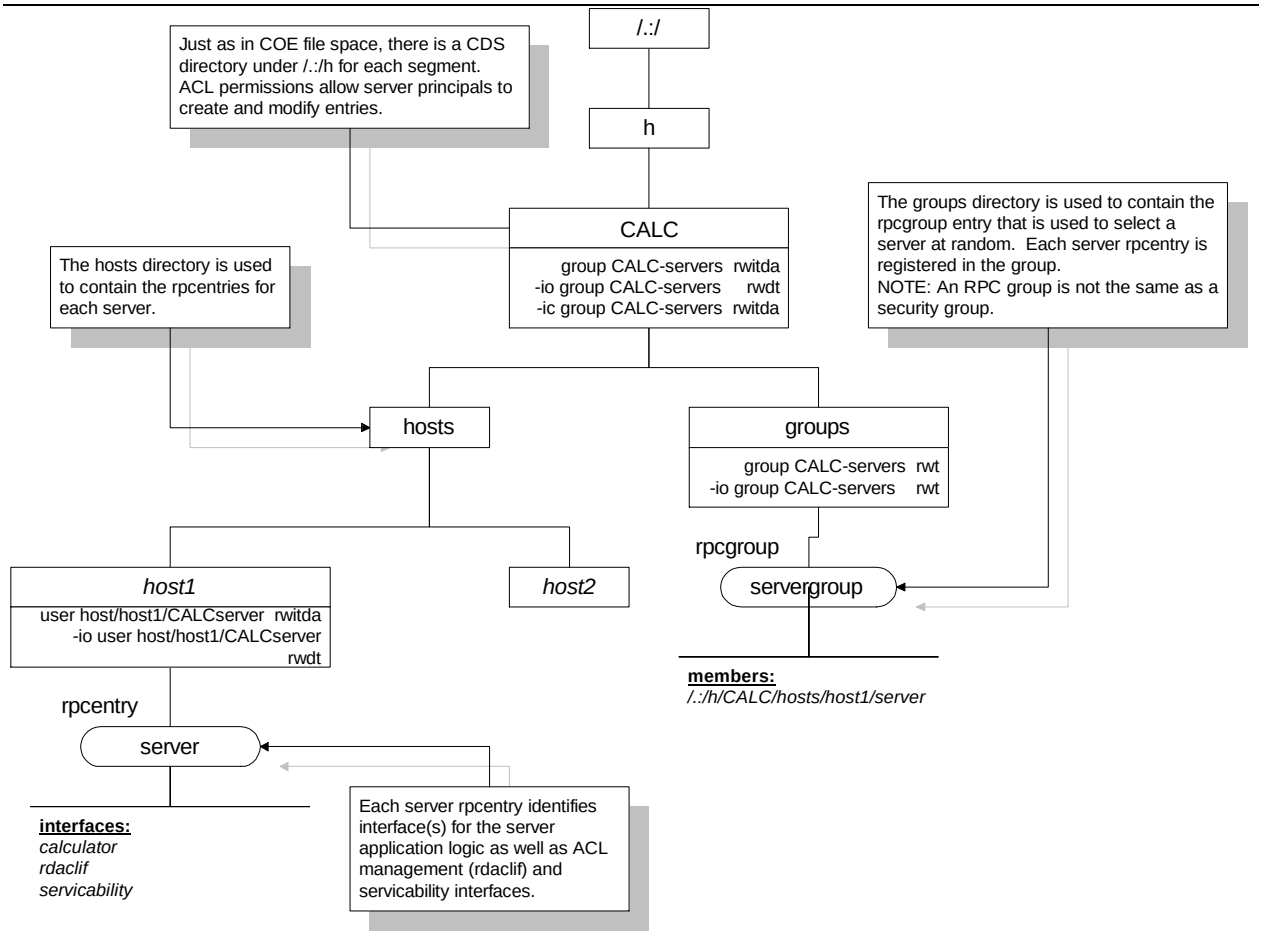

Figure 8 Sample Application CDS Namespace

This is a suggested CDS organization in accordance with the recommendations in the I&RTS Appendix X. This structure is not suitable for all purposes. For example, if there is a server installed on every host, this structure could create a large number of host directory entries. In some cases it is more efficient to group servers by function rather than host name. This structure is used by the sample application and its installation scripts. However it is not enforced by the DCECOE library. The only requirement is that the client srvrconf entryname attribute point to an rpcgroup or rpcprofile in CDS as a starting point for the search for a suitable server. Chapter 4 contains additional information on the use of attributes. The use of **/.:/h/SEGMENTNAME** to organize CDS is strongly encouraged.

3.7.2 Security Registry

This section illustrates the DCE principals and groups used in the sample CALC application.

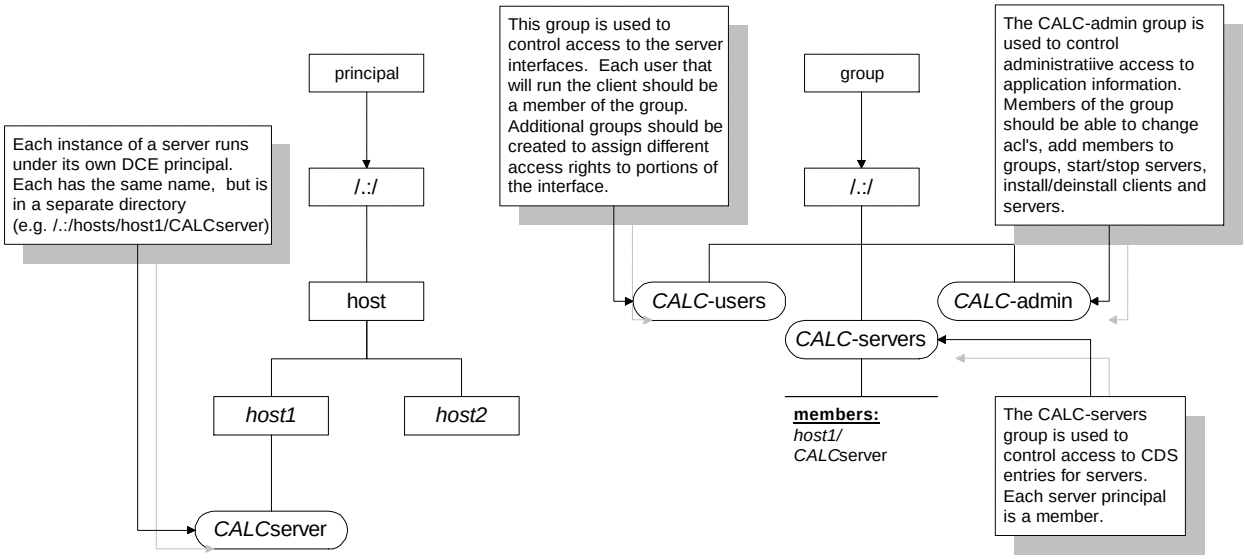

Figure 9 Sample Application Security Registry

By convention there is a separate server principal for each server instance so that audit records precisely identify the originator of all actions. The installation scripts for the sample calculator application follow this convention. However, this is not required by the DCECOE library. The only requirement is that the principals attribute in the server configuration record contain a valid principal to use in running the server. Chapter 4 contains additional information about the use of attributes.

Also by convention, the installation scripts for the calculator application create three groups. The **CALC-users** group contains users who are allowed to run clients that access the server. The **CALC-admin** group contains users who are allowed to administer the application, including modifying ACLs for the server, assigning users to CALC groups, or starting/stopping CALC servers. The **CALC-servers** group contains all the server principals.

Additional groups may be needed for specific applications. For example, a **CALC-adders** group could be created, along with suitable ACLs within CDS, containing users who are allowed to perform the add operation but not the subtract operation.

3.7.3 Host Table

This section illustrates the use of srvconf and xattrschema.

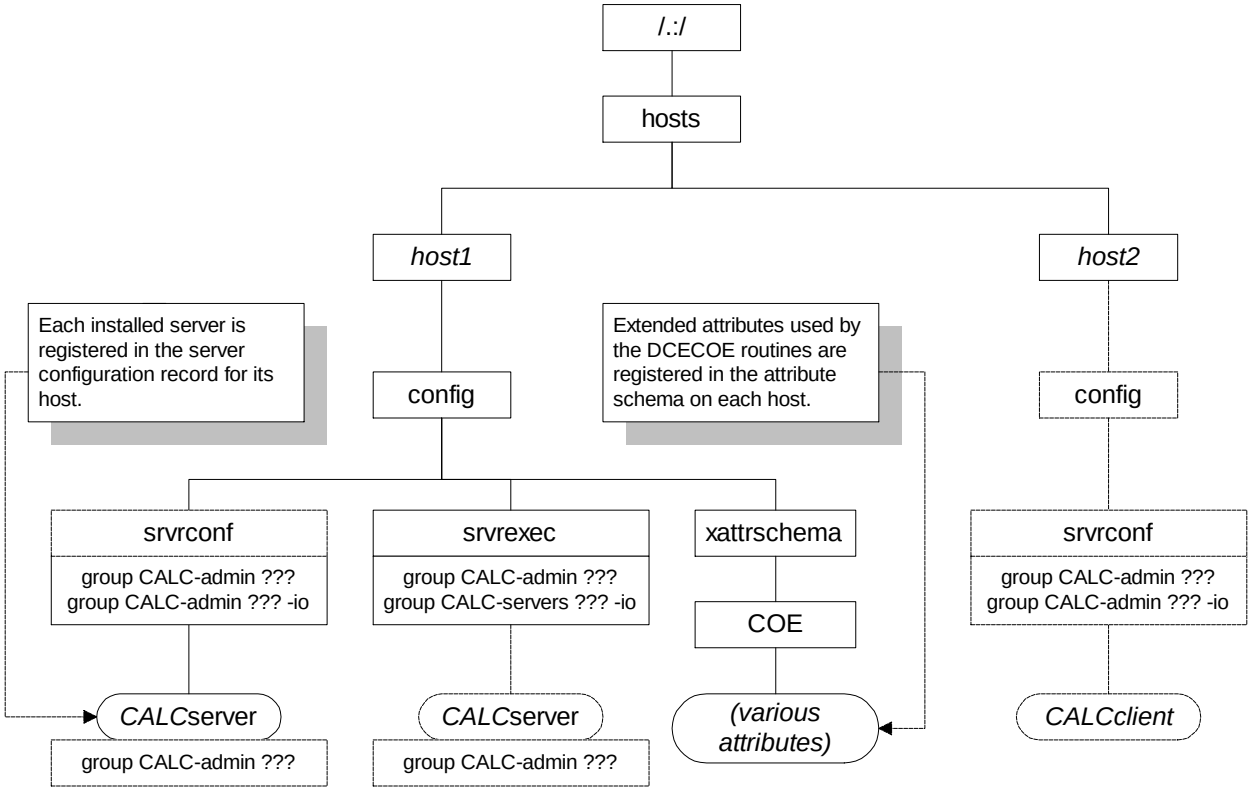

Figure 10 Sample Application Host Namespace

4. DCECOE Attributes

4.1 Use of the DCED

The **dced** runs on every DCE machine and provides a set of host-specific services for managing DCE applications on those systems. In particular, the DCE supports a set of five databases which are all ACL-secured and are appropriately junctions into the DCE namespace. The DCECOE makes extensive use of all of these services.

srvrconf - server configuration	Maintains a set of extensible records which describe a server Used to start instances of servers Servers use these records as initialization and configuration parameters
srvrexec - server execution	Maintains a record of a running server Used to stop instances of servers
hostdata - host file services	Provides remote access to local system files Used for performing remote configuration and reporting. Currently only used to provide remote access to local audit files
keytab - key table services	Provides remote access to server key tab files Used during installation, password maintenance, and backup
xattrschema - extended attribute schema definition	Used as schema database to describe "extended attributes" in the srvrconf database

Table 1 Host Specific Databases

4.2 Example Server Configuration Attributes

The following is an example of the **srvrconf** information for the sample application, as printed using the command **`dcecp -c server show CALCserver`**. The extended attributes used by the DCECOE library are identified with a **COE/** prefix. The remaining attributes are standard server attributes used by **dced**. For further information see the **xattrschema(8rpc)** manual page.

```
{uuid 4222ef4c-365c-11d0-8016-ccfc0d7baa77}
{program CALCserver}
{arguments {}}
{prerequisites {}}
{keytabs 418bd36e-365c-11d0-8016-ccfc0d7baa77}
{entryname {}}
{services
 {{ifname rdaclif}
```

```
    {annotation {Standard ACL interface}}
    {interface {47b33331-8000-0000-0d00-01dc6c000000 1.0}}
    {bindings {}}
    {objects 01eb03d6-e2ee-11cf-91f9-ce9cdd02aa77}
    {flags {}}
    {entryname /.../gccs.smil.mil/h/CALC/hosts/borg/server}}
   {{ifname calculator}
    {annotation {Basic calculator application}}
    {interface {0073a028-fbdb-1e53-908e-08002b13ca26 1.0}}
    {bindings {}}
    {objects {}}
    {flags {}}
    {entryname /.../gccs.smil.mil/h/CALC/hosts/borg/server}}
   {{ifname serviceability}
    {annotation {DCE Serviceability}}
  {interface {000cf72e-0688-1acb-97ad-08002b12b8f8 1.0}}
    {bindings {}}
    {objects 01eb03d6-e2ee-11cf-91f9-ce9cdd02aa77}
    {flags {}}
    {entryname /.../gccs.smil.mil/h/CALC/hosts/borg/server}}}
  {principals /.../gccs.smil.mil/hosts/borg/CALCserver}
  {starton {}}
  {uid 500}
  {gid 1}
  {dir /h/CALC/bin}
  {COE/DebugService coe:*.9:TEXTFILE:/h/CALC/data/server.out}
  {COE/ServerThreads 5}
  {COE/AclMgrType aclobject,flat}
  {COE/AclMgrUuid 6ba40bf6-e2ee-11cf-8d13-ce9cdd02aa77}
  {COE/AclMgrInfo Calculator {Sample Calculator Refmon}}
  {COE/AclMgrDesc c control 8}
  {COE/AclMgrDesc t test 64}
  {COE/AclMgrDesc a add 128}
  {COE/AclMgrDesc s subtract 256}
  {COE/AclMgrDefault group subsys/dce/dced-admin ct}
  {COE/AclMgrDefault group CALC-admin ct}
  {COE/AclMgrDefault user hosts/borg/CALCserver asct}
  {COE/AuditFirst 281587713}
  {COE/AuditEvents 2}
  {COE/AuditMsgs cal}
  {COE/MgmtMapping tttta}
  {COE/Service WARNING:TEXTFILE:/h/CALC/data/warning.log}
  {COE/DcecpOp
    {rpcgroup add /.../gccs.smil.mil/h/CALC/groups/servergroup
      -member /.../gccs.smil.mil/h/CALC/hosts/borg/server}}
  {COE/KeytabFile /h/CALC/data/keytab/CALC.tab}
  {COE/AuditTrail /h/CALC/data/audit.aud}
  {COE/AclSetup /h/CALC/bin/CALCaclsetup}
  {COE/AclFile /h/CALC/data/CALC}
  {COE/AclNameFile /h/CALC/data/CALCname}
```

```
{COE/ClientBind {{dce hosts/borg/CALCserver default default dce}
  /.../gccs.smil.mil/h/CALC/hosts/borg/server}}
```

The information in this record can be divided into two categories: DCE defined attributes (such as program, directory, uid, services) and DCECOE defined extended attributes. The DCECOE attributes are all catalogued in the extended attribute database of each **dced**.

The trick of using the DCECOE library is to correctly configure these records. All detailed activity is performed following the settings in the repository.

A note about repository attributes. Standard DCE attributes cannot be changed once created. In order to change a DCE attribute, the record must be deleted and recreated. However DCECOE attributes can be changed at any time, and will normally take effect the next time the server is initialized. All changes to configuration records are controlled by ACLs in the **dced**. The **dced** repository can be edited from anywhere (thanks to CDS and RPC) facilitating remote management, troubleshooting, verification and configuration management. Of the server initialization flags, certain flags require the presence of attributes in the configuration record. The manual pages describe which attributes are required for which flags.

4.3 DCECOE Attributes

DCE extended attributes in the extended attribute registry do not define data; they only describe the data that can be instantiated (i.e. placed on a srvrconf record). The data types supported are: printstring, stringarray, int, bytes, uuid, and binding. Each attribute can be further classified as being single value or multi-value. This controls the number of times that the template can be instantiated on a single object. The following table describes the predefined COEDCE attributes. The application writer is encouraged to use attributes and interfaces exist to query/modify these values administratively and programatically. The following table lists the COEDCE attributes, which are described in following paragraphs.

Name	Multi-value	Type	Format	Description
SvcTableName	yes	printstring	3 chars	Serviceability table names
MgmtMapping	no	printstring	5 chars	Management function mapping to ACL
MgmtAcl	no	uuid	uuid	UUID of management ACL
MgmtAclMgr	no	uuid	uuid	Manager type uuid of management ACL
AclFile	no	printstring	path/file name	File to save ACL database
AclNameFile	no	printstring	path/file name	File to save ACL name translation
AclSetup	no	printstring	path/file name	ACL initialization script file

Name	Multi-value	Type	Format	Description
AclMgrUuid	no	uuid	uuid	ACL manager type uuid
AclMgrInfo	no	stringarray	2 elements: name, help	Information identifying the ACL manager (name and annotation)
AclMgrDesc	yes	stringarray	3 elements: name, description, decimal value	Permissions (character name, title, and decimal value)
AclMgrDefault	yes	stringarray	3 elements: type, key, permissions	Default ACLs for server objects ('group' or 'user', identity of group or user, permissions)
AclMgrType	no	printstring	Comma separated: aclobject, defcontainer. defobject name, uuid	ACL Database characteristics
AuditTrail	no	printstring	path/file name or 'central'	File name or 'central'
AuditFirst	no	integer	decimal value	First decimal event number
AuditEvents	no	integer	decimal	Number of events
AuditMsgs	yes	printstring	3 char.	Message component for events
AuditClasses	yes	printstring	characters	Defined classes, filenames
ServerThreads	no	integer	decimal	Max. number of server threads
DcecpOp	yes	stringarray	dcecp command	dcecp commands - i.e. for CDS, profiles
Service	yes	printstring	type:destination: name	Serviceability setting
DebugService	yes	printstring	component: level.level: destination:nam e	Debug Serviceability setting
KeytabFile	no	printstring	path/file name	Filename of keytab file
ClientBind	no	binding	See binding in xattrschema man page	Client binding requirements - used by server

Table 2 Predefined COEDCE Attributes

4.4 Using attributes

The following paragraphs describe the extended attributes used by the DCECOE library routines.

4.4.1 SvcTableName

This attribute is used to list the serviceability component names used by this application. The DCECOE library uses the **coe** component, but your application may use others. This attribute is intended to be used during verification to detect the appropriate message catalogs are installed and to ensure that conflicting components are not installed. A component name must consist of 3 lower-case characters.

4.4.2 MgmtMapping

This set of attributes is used to control and configure the management functions that all DCE applications support. Management functions allow a client to request interface information, server principal name, or statistics from the server, to ping the server, or to stop the server. For a further discussion, see the **rpc_mgmt_set_authorization_fn(3)** man page. There are five management operations and the **Mgmtmapping** attribute defines the relationship between permissions understood by the ACL manager/Reference monitor permissions. The attribute defines the permissions that must be present to allow the client to perform the management function. If this attribute is missing, the DCECOE library assumes '**ttttc**' representing the standard 'test' and 'control' permissions. The ACL to be checked is attached to the **srvrexec** object for the server.

4.4.3 MgmtAcl

This attribute defines the UUID of the ACL object which will be used as the permission settings associated with management operations. If this parameter is missing, the DCECOE creates a default ACL object named **mgmt** under the server's rpcentry (e.g. **/.:/h/CALC/hosts/borg/server/mgmt**.

4.4.4 MgmtAclMgr

This is the UUID of the ACL manager which imposes semantics over the management ACL. (i.e. the routine that provides a reference monitor to test permissions to the management functions. If not supplied, DCECOE uses the ACL manager UUID associated with the standard ACL manager supplied with DCECOE.

Note: none of the Mgmt attributes are used, if the caller supplies the 'mgmt_function' callback during **COEDCEinitialize_server()**.

4.4.5 AclFile and AclNameFile

The **AclFile** and **AclNameFile** attributes define the names of the files used to house the default ACL database and its ancillary name-to-ACL database. If their values do not specify absolute

pathnames, the files will be stored under **/h/SEGMENT/data**. A "**.db**" extension is added to the names provided. These attributes are mandatory when using the **S_ACL** flag.

4.4.6 AclSetup

The **AclSetup** parameter defines the name of an executable file (usually a **dcecp** script) which will be run to initialize the ACL database. This script is run as a side effect of server initialization whenever the ACL database is not present. It is a natural extension to server installation and configuration, but is delayed until the first time the server starts because the server must be running in order to perform ACL updates. When the DCECOE recognizes that the ACL databases are empty, it runs this program, which by default is the **/h/SEGMENT/bin/aclsetup** script. This script performs a configurable set of **acl show** and **acl modify dcecp** commands which can be used to modify initial ACLs to any configuration without hard-coding this in the application.

Two procedures are provided as part of the default acl manager to simplify ACL setup. The command **newacl** is used to create new ACLs and **setacl** is used to modify the ACL. Here is an extract from the default **aclsetup** program.

```
#/usr/bin/dcecp
/* support routines skipped */
getbind [lindex $argv 0]

newacl calculator
setacl calculator add {group acct-admin asc}
```

When **newacl** is used, DCECOE uses the values of the **AclMgrDefault** attribute to give the ACL an initial set of values. This attribute is multi-valued and can contain any combination of 'group' or 'user' ACL entries.

Every ACL manager defines a UUID which represents a set of permissions supported by the ACL manager. The **AclMgrUUID** attribute allows the user to define this UUID. This attribute must be present in order to use the **S_ACL** attribute. The value may be assigned using **uuid_create(3)** or more often by **uuidgen(1)**.

4.4.7 AclMgrInfo

The **AclMgrInfo** attribute represents the string information including the ACL Managers name and description. This attribute is mandatory when using the **S_ACL** flag.

4.4.8 AclMgrDesc

The **AclMgrDesc** attribute is a multi-valued string array, with each array entry consisting of three elements; a name, description, and decimal value. Each part of the array represents a permission

bit that the ACL manager implements. The are several ACL bit permissions that are recommended by OSF, listed in the table below. These are defined in **<dce/aclbase.h>** . To avoid confusion on the part of administrators, these values should be used whenever they are applicable.

Permission	Value
read	1
write	2
execute	4
control	8
insert	16
delete	32
test	64

4.4.9 AclMgrType

The **AclMgrType** attribute is reserved to define the structure and type of the ACL Manager. It consists of a string which can contain one or more of the supported object types and one of the structure types:

The following object types have been defined:

aclobject - supports ACLs on simple objects
defobject - supports default inheritance ACLs on objects
defcontainer - supports default inheritance ACLs on containers

The following structural attributes are defined:

flat - the database contains no hierarchical structure
hier - the database supports full hierarchy (e.g. a filesystem)
bilevel - the database does not support containers within containers
sparse - the database supports sparse searching
noleaf - the database permits hierarchy but only as a side effect of creating a leaf

Note: only the 'flat', 'bilevel', and 'hier' structure are currently supported.

4.4.10 AuditTrail

The **AuditTrail** attribute represents the filename into which audit records will be created. The special name "central" is used when the client wishes to use the central audit trail supported by the audit daemon. The use of the central audit file is strongly encouraged.

4.4.11 AuditFirst

The **AuditFirst** attribute provides the numeric value representing the first audit event. In DCECOE a SAMS file is used to create DCE messages which will be used for audit events. The first message number is placed in **AuditFirst** and the number of events are placed in AuditEvents. The name of the SAMS component is placed in **AuditMsgs**. See the *Application Development Guide - Core Components* Chapter 3 for more information.

Here is an example **calc.sams** file for the sample application:

```
# Part I
# This part defines the lowest-level table, the one that
contains
#  the messages (defined in the third part) in a straight array.
component       cal
table           cal__table
technology      dce


# Part II
# This part defines the sub-component table, each element of
#  which contains the base address of one of the sub-component
#  message tables.
serviceability  table cal_svc_table handle cal_svc_handle
start
    sub-component cal_s_manager "manager"  cal_i_svc_manager
    sub-component cal_s_server  "server"   cal_i_svc_server
end

#
# Part IIa
# This part contains event codes for auditing
#
start
code  add_event
text  "add operation"
action ""
explanation ""
end

start
code  subtract_event
text  "subtract operation"
action ""
explanation ""
end

# Part III
```

```
# This part defines the serviceability messages.
#
```

The value of **AuditFirst** must match the value of the first application-defined event, in this case **add_event**, from the **dcecalmsg.h** file (**smallest_cal_message_id** or **add_event**). The following is an example of the **dcecalmsg.h** generated from the sams file above.

```
/* Generated from calc.sams on 1996-07-21-11:31:40.000 */
/* Do not edit! */
#if  !defined(_DCE_DCECALMSG_)
#define _DCE_DCECALMSG_
#define add_event                          0x10c8b001
#define subtract_event                     0x10c8b002
#define cal_sad_ending                     0x10c8b003
#define cal_i_svc_manager                  0x10c8b004
#define cal_i_svc_server                   0x10c8b005

#define smallest_cal_message_id            0x10c8b001
#define biggest_cal_message_id             0x10c8b005

#endif    /* !defined(_DCE_DCECALMSG_) */
```

4.4.12 AuditClasses

The **AuditClasses** attribute is used to catalog the event classes which are distributed with this server. An audit class is a file used to facilitate the administration of audit filters. See the *OSF DCE Administrators Guide - Core Components* for more information about audit classes and filters.

4.4.13 ServerThreads

The **ServerThreads** attribute represents the number of call threads that the DCE runtime creates in order to service incoming RPC requests. This value is used in the **rpc_server_listen()** call when the **S_LISTEN** flag is used. Use the value of 1 if the server's manager functions are not capable of being multi-threaded.

4.4.14 DCEop

The **DCEop** attribute is used to collect operations to be performed when the server initializes. These are typically **dcecp** commands which would be difficult to perform using programming APIs. An example might be creating a CDS directory. This is a multi-valued attribute and will be invoked by the server by executing **dcecp** as follows:

```
dcecp -c DCEop1; DCEop2
```

4.4.15 Service and DebugService

The **Service** and **DebugService** attributes are used to set serviceability options. Please refer to the **svc_route(5)** man page for an explanation of using these serviceability settings.
The **Service** entry consists of three fields specify routing for non-debug serviceability messages. The format is as follows:

```
sev:out_form:dest[;out_form:dest . . . ] [GOESTO:{sev | comp}]
```

The **sev** (severity) field specifies one of the following message severities: **FATAL, ERROR, WARNING, NOTICE, NOTICE_VERBOSE**. The **out_form** (output form) field specifies how the messages of a given severity level should be processed, and must be one of the following**: BINFILE, TEXTFILE, FILE, DISCARD, STDOUT, STDERR**. The out_form specifier may be followed by a two-number specifier of the form: **.gens.count** where: **gens** is an integer that specifies the number of files (i.e., generations) that should be kept and **count** is an integer specifying how many entries (i.e., messages) should be written to each file. The **dest** (destination) field specifies where the message should be sent, and is a pathname. The field can be left blank if the out_form specified is DISCARD, STDOUT, or STDERR. The field can also contain a %ld string in the filename which, when the file is written, will be replaced by the process ID of the program that wrote the message(s). Filenames may not contain colons or periods.

The format for the **DebugService** routing specifier string is:

```
component:sub_comp.level,...:out_form:dest[;out_form:dest...]
  [GOESTO:{sev | component}]
```

Where **out_form**, **dest**, and **sev** have the same meanings as defined earlier in this reference page. Nine serviceability debug message levels (specified respectively by single digits from 1 to 9) are available. The precise meaning of each level varies with the application or DCE component in question, but the general notion is that ascending to a higher level (for example, from 2 to 3) increases the level of informational detail in the messages. Setting debug messaging at a certain level means that all levels up to and including the specified level are enabled. The general format for the debug level specifier string is:

```
component:sub_comp.level,sub_comp.level,. . .
```

Where: **component** is the three-character serviceability component code for the program whose debug message levels are being specified, **sub_comp.level** is a serviceability subcomponent name, followed (after a dot) by a debug level (expressed as a single digit from 1 to 9). Note that multiple subcomponent/level pairs can be specified in the string. If there are multiple

subcomponents and it is desired to set the debug level to be the same for all of them, then the form: component:*.level will do this (where the ``*" specifies all subcomponents).

4.4.16 KeytabFile

The **KeytabFile** records the name of the keytab file and is used when access to the keytab database is not accessible. It should match the **storage** attribute of the matching keytab object.

4.4.17 ClientBind

The **ClientBind** attribute is read by the client in order to determine what security settings to use in contacting the server. The value of this attribute contains authentication, authorization and binding information suitable for communicating with a DCE server. The syntax is a list of two elements. The first element is a list of security information where the first element is the authentication type, either **none** or **dce**, followed by information specific for each type. The type **none** has no further info. The type **dce** is followed by a principal name, a protection level (one of **default**, **none**, **connect**, **call**, **pkt**, **pktinteg**, or **pktprivacy**), an authentication service (one of **default**, **none**, or **secret**), and an authorization service (one of **none**, **name**, or **dce**). Examples of three security information lists are:

```
{none}
{dce /.:/melman default default dce}
{dce /.:/melman pktprivacy secret dce}
```

The second element is a list of binding information, where binding information can be string bindings or server entry names. Two examples of binding information are:

```
{/.:/hosts/hostname/dce-entity /.:/subsys/dce/sec/master}
{ncadg_udp_ip:130.105.96.3[123] ncadg_udp_ip:130.105.96.6[123]}
```

The values are obtained by the client and used to establish the security environment for remote procedure calls using **rpc_binding_set_auth_info(3rpc)**. For COE applications, the suggested entry is as shown in the sample earlier, using defaults for protection and authentication services. Refer to the **xattrschema(5rpc)** man page for additional information.

4.5 Other interfaces for accessing attributes

No DCECOE interface exists for a server to obtain a list of attributes. See the standard OSF DCE **dce_inquire_server()** and **dce_read_server()** interfaces.

Attributes are represented by the **sec_attr_t** data structure defined in **<dce/sec_attr_base.h>**. A useful set of macros is available in **<dce/sec_attr_tools.h>.**

4.6 Client configuration objects

When a client is installed, a configuration record is also created. This record is not used to start the client, but rather to describe which services the client depends on. The only extended attributes that are applicable in the client record are the **Service**, **DebugService**, and **KeytabFile**.

The most critical standard attributes for clients are the **services** attribute(s). There should be one entry for each server interface used by the application. The **interface** defines the UUID of the server interface, and must match the one in the server interface of the same name. The **entryname** attribute contains the location of an rpcgroup or rpcprofile in CDS to begin the search for a server. For the sample application, the search starts at **/.:/h/CALC/groups/servergroup**. Note that this starting point may be anywhere in CDS, and can be configured differently for different client machines within a cell. This allows different machines to have configurable search paths for servers.

The following displays the configuration entries for the sample client application:

```
      {uuid 4c31d4da-365c-11d0-9ac8-ccfc0d7baa77}
      {program CALCserver}
      {arguments {}}
      {prerequisites {}}
      {keytabs {}}
      {entryname {}}
      {services
       {{ifname CALC}
        {annotation {Basic calculator application}}
        {interface {0073a028-fbdb-1e53-908e-08002b13ca26 1.0}}
        {bindings {}}
        {objects {}}
        {flags {}}
        {entryname /.../gccs.smil.mil/h/CALC/groups/servergroup}}}
      {principals {}}
      {starton {}}
      {uid 0}
      {gid 0}
{dir /h/CALC/bin}
```

APPENDIX A - DCECOE MANUAL PAGES

This appendix provides a complete set of man reference pages for the DCECOE library. These man pages are provided on-line in the ./DCE_API/man directory.

COEDCE(3rpc)

NAME

COEDCE COE-unique DCE RPC library

SYNOPSIS

#include <dcecoe/dcecoe.h>

DESCRIPTION

Functions in the DCECOE RPC library provide a simplified mechanism for developing client-server applications using the OSF DCE services. All of the procedure calls required to initialize a DCE client or server are consolidated into a single procedure call. The routines make use of sensible, but overridable, defaults to simplify the development of applications. The DCECOE library takes advantage of and provides easy access to many of the features of OSF DCE Version 1.1, as described below.

Functions

The DCECOE library consists of the routines listed below and described in separate man pages.

Server-side routines:

COEDCEinitialize_server()	Initializes a DCE server.
COEDCEsignal_server()	Signal a server to enter listen loop.
COEDCEcreate_acl()	Creates an access control list (ACL).
COEDCEis_auth()	Makes an authorization decision.
COEDCEfinalize_server()	Terminate server resources.

Client-side routines:

COEDCEinitialize_client()	Initializes a DCE client
COEDCElocate_server()	Locates a server
COEDCEgetvector()	Retrieves a binding vector
COEDCEinquire_server()	Gets info about a server
COEDCEstart_server()	Prepare a handle for communications with a server
COEDCEfree_servers()	Frees a server
COEDCEfinalize_client()	Frees allocated resources

Features

Security

The security of an RPC connection can be configured to any level from unauthenticated to authenticated and encrypted. The DCECOE routines automatically perform server login,

authentication, password maintenance, and security context refresh. They guarantee that clients use appropriate security choices as required by servers. DCE security mechanisms are used to identify and authenticate servers rather than inquiring of servers themselves for security identification.

ACL Management/Reference Monitor

The server routines include an access control list (ACL) manager to allow remote management of ACL's for the server. An application may define its own functions to be controlled (e.g. read/write/delete for storage, print/control for a printer, view/update for document, etc.). At built-in reference monitor makes access decisions based on the contents of the ACL database and performs access auditing.

Server Administration

The DCECOE server library implements a management interface that allows the server to be remotely managed using the standard dcecp file locations, and application configurables, are maintained as extended attributes within CDS.

Server Startup

The DCECOE client library provides functions to allow a server to be started on demand if one is not currently running.

Server Connection and Query

The client library allows the client to connect to any available server, or to locate all available servers and retrieve information about the servers in order to make a connection decision. Decisions can be made based on the availability of the server, the objects' maintained by the server, or any other information agreed upon between the client and server and recorded in the configuration information within CDS.

Auditing

The server routines maintain an audit file that can be written by the reference monitor or the application using standard OSF DCE audit functions.

Serviceability Messages

The DCECOE library functions make use of the OSF DCE 1.1 serviceability interfaces to generate and manage error messages. The server management interface allows messages of different severity to be turned on or off and routed to different locations (e.g. error log, stderr, etc.).

FILES

/usr/include/dcecoe/dcecoe.h
/usr/lib/libdcecoe.a
/opt/dcelocal/nls/msg/dcecoe.cat

SEE ALSO

DCECOEcreate_acl(3dce)
DCECOEinitialize_server(3dce)
DCECOEinitialize_client(3dce)
DCECOEis_auth(3dc
DCECOEfinalize_server(3dce)
DCECOEfinalize_client(3dce)
DCECOElocate_server(3dce)
DCECOEgetvector(3dce)
DCECOEinquire_server(3dce)
DCECOEfree_servers(3dce)
DCECOEsignal_server(3dce)
OSF DCE Application Development Guide - Core Components

NOTES

None

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

COEDCEcreate_acl(3rpc)

NAME

COEDCEcreate_acl - Creates an access control list (ACL)

SYNOPSIS

```
#include <dcecoe.h>

error_status_t COEDCEcreate_acl(object_p_t objp);
```

PARAMETERS

objp

A pointer to an object structure describing the new object

DESCRIPTION

The COEDCEcreate_acl() routine creates a new ACL object. It is used by an server which manages dynamic objects (such as a file system). It is intended to be called from a successful create object operation (such as open(), creat()). The other way to create ACLs is during ACL initialization (see aclsetup).

RETURN VALUES

dce_error - a DCE error is responsible for failure, see dce_error_inq_text(3)

SEE ALSO

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

COEDCEfinalize_client(3rpc)

NAME

COEDCEfinalize_client - Frees allocated resources

SYNOPSIS

```
#include <dcecoe.h>

void COEDCEfinalize_client(
        unsigned32          flags,
        error_status_t *status);
```

PARAMETERS

flags

A set combinable option flags.

status

A pointer to a variable to hold a return status.

DESCRIPTION

The COEDCEfinalize_client() routine frees the resources belonging to the client. It is made by a client application prior to application termination. This call is not necessary unless COEDCEinitialize_client() returns successfully.

FLAGS

C_LOGOUT

Performs a DCE logout and destroys the associated credentials file. This option will remove your current credentials if you did not use C_LOGIN in COEDCEinitialize_client().

RETURN VALUES

No value is returned.

SEE ALSO

COEDCEinitialize_client()

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

COEDCEfinalize_server(3rpc)

NAME

COEDCEfinalize_server - Terminate resources

SYNOPSIS

```
#include <dcecoe.h>

void COEDCEfinalize_server();
```

PARAMETERS

None

DESCRIPTION

The COEDCEfinalize_server() routine terminates any resources obtained for the server during COEDCEinitialize_server(). This routine need not be used unless COEDCEinitialize_server() completes successfully.

SEE ALSO

COEDCEinitialize_server()

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

COEDCEfree_servers(3rpc)

NAME

COEDCEfree_servers - Frees a server

SYNOPSIS

```
#include <dcecoe.h>

unsigned32 COEDCEfree_servers(
        server_t        *servers,
        unsigned32      count,
        error_status_t *status);
```

PARAMETERS

servers

A pointer holding allocated server structures returned by a successful call to COEDCEinquire_server() client's configuration record. *servers* can point to one or more server_t structures depending on the count supplied to COEDCEinquire_server().

count

The number of server structures pointer to by `servers' parameter.

status

A pointer to a variable used to hold the DCE return status.

DESCRIPTION

The COEDCEfree_servers() routine frees any server_t structures and associated resources passed to it. This function should be used after the server_t structures are no longer needed.

RETURN VALUES

bad_parameter - API arguments are malformed

SEE ALSO

COEDCEinquire_server()

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

COEDCEgetvector(3rpc)

NAME

COEDCEgetvector - Retrieves a binding vector

SYNOPSIS

```
#include <dcecoe.h>

rpc_binding_handle_t COEDCEgetvector();
```

PARAMETERS

None

DESCRIPTION

The COEDCEgetvector() routine retrieves a binding vector obtained using COEDCElocate_server().

RETURN VALUES

Returns a pointer to a vector, or NULL if none was found.

SEE ALSO

<dce/rpcbase.h> contains the definition for the rpc_binding_vector_t. COEDCElocate_server() is used to obtain the binding vector.

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

COEDCEinitialize_client(3rpc)

NAME

COEDCEinitialize_client - Initializes a client

SYNOPSIS

```
#include <dcecoe.h>

unsigned32 COEDCEinitialize_client(
        char            *segmentname,
        unsigned32      flags,
        error_status_t *status);
```

PARAMETERS

segmentname

A string containing the name of the SEGMENT. This name must conform to the COE/DCE naming standards and consist of 4 uppercase characters. The name supplies is postpended with ``client" for form the name of the requisite DCED srvrconf configuration record.

flags

A set of options which can be combined to produce a variety of client behaviors. See the description for restrictions and a list of mandatory extended attributes required for each flag.

status

A pointer to a variable to hold the DCE return status.

DESCRIPTION

The COEDCEinitialize_client() routine reads the client's configuration record and performs initialization of DCE serviceability, DCE messaging, and non-interactive login and context refresh depending on the C_LOGIN and C_REFRESH flags.

FLAGS

C_LOGIN

Performs a DCE login using a keytab file. Required DCE Attributes: keytabs, principals Required Extended Attributes: KeytabFile

C_REFRESH

Perform login refresh (assumes login via a key file). Available only with the C_LOGIN flag.

ADDITIONAL ATTRIBUTES

Service

production level serviceability settings

DebugService

debug level serviceability settings

RETURN VALUES

bad_parameter - request arguments are malformed
bad_configuration - local DCED configuration record is missing or unusable
bad_flags - invalid or conflicting flags
dce_error - a DCE error is responsible for failure, see dce_error_inq_text(3)

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

COEDCEinitialize_server(3rpc)

NAME

COEDCEinitialize_server - Initializes a DCE server

SYNOPSIS

```
#include <dcecoe.h>

unsigned32 COEDCEinitialize_server(
        unsigned char *segmentname,
        unsigned32          flags,
        callbacks_p_t callbacks,
        error_status_t *status);
```

PARAMETERS

segmentname

The segmentname of the server's SEGMENT (4 uppercase characters). This argument is postpended with ``server'' to form the configuration record name in the DCED srvrconf database.

flags

A set of options which modify the behaviors of the server

callbacks

A pointer to a structure defining optional callbacks used for further customization of the DCE server

status

A pointer to a variable to hold a DCE return status code.

DESCRIPTION

The COEDCEinitialize_server() routine performs comprehensive server initialization for a DCE server. It is controlled largely by a set of attributes placed in the DCED configuration repository and can be further customized using an array of callback functions.

FLAGS

S_LOGIN

Perform a DCE login using a key file. Required DCE Attributes: keytabs, principals Required Extended Attributes: KeytabFile

S_REFRESH

Perform login refresh (assumes login via a key file).

S_AUDIT

Initializes the server for auditing Required Attributes: AuditTrail, AuditFirst, AuditEvents, AuditMsgs, AuditClasses

S_HDATA

Creates the `audit' hostdata entry for remote access to the audit trail file.

S_ACL

Initialize the server's ACL management/reference monitor Required Attributes: AclSetup, AclMgrUuid, AclMgrInfo, AclMgrDesc, AclMgrDefault, AclMgrType, DefaultAcl Optional callbacks: Use the `objclassfunc' callback to provide object/container mapping for a multi-level ACL manager

S_LISTEN

Perform a rpc_server_listen() rather than returning. Return after the server stops listening.

S_WAIT

Wait until signaled via a condition variable before doing the rpc_server_listen(). See COEDCEsignal_server() for signaling.

S_CLEANUP

Perform full cleanup after returning from rpc_server_listen().

S_KEYMGMT

Perform DCE key management (i.e. changing of passwords as required by the Registry properties).

S_MGMTAUTH

Initialize the server's management authorization function. Required Attributes: MgmtMapping, MgmgAcl, MgmtAclMgr Optional callbacks: Use the `mgmtauth' callback to register a user defined callback function

S_CDSEXPORT

Export information into CDS as configured in the srvconf record.

ADDITIONAL ATTRIBUTES

Service
production serviceability settings

DebugService
debug serviceability settings

DcecpOp
run the following dcecp operations during startup

ServerThreads
allow a maximum concurrency value (call threads in rpc_server_listen)

RETURN VALUES

bad_configuration - local DCED configuration record is missing or unusable

dce_error - a DCE error is responsible for failure, see dce_error_inq_text(3)

SEE ALSO

COEDCEsignal_server() - used to resume a waiting server (S_WAIT)
COEDCEserver_finalize() - used to perform cleanup when S_CLEANUP is not used

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

COEDCEinquire_server(3rpc)

NAME

COEDCEinquire_server - Gets info about a server

SYNOPSIS

```
#include <dcecoe.h>

unsigned32 COEDCEinquire_server(
        unsigned32              flags,
        unsigned32              service,
        unsigned32              *count,
        server_t                **servers,
        rpc_binding_handle_t object,
        error_status_t          *status);
```

PARAMETERS

flags

A set of combinable option flags which determine the type of information returned about the server

service

The index of the service description in the client's configuration record, about which information is being requested

count

A pointer used to indicate the maximum number of server instances about which data is requested

servers

pointer to a pointer used to hold allocated structures by the routine if information was successful

handle

An RPC binding handle indicating the server host to query

status

A pointer to a variable to hold a return status.

DESCRIPTION

The COEDCEinquire_server() routine retrieves configuration information about defined or running servers. This call uses the binding handle to contact the DCED to inquire either the srvrconf or srvrexec databases.

Use the COEDCEfree_servers() call to return allocated storage.

FLAGS

Only one of the following flags can be used.

C_CONF

Retrieve information about a single configured server. This queries the configuration record (template) maintained for the server.

C_EXEC

Retrieve information about `count' running servers. This returns execution state information as well as the configuration data.

RETURN VALUES

bad_parameter - request arguments are malformed

bad_configuration - local DCED configuration record is missing or unusable

bad_flags - invalid or conflicting flags

dce_error - a DCE error is responsible for failure, see dce_error_inq_text(3)

SEE ALSO

COEDCElocate_server() is used to obtain binding vector COEDCEfree_servers() is used to free server structures returned by COEDCEinquire_server() <dce/dced_base.h> - the DCE header containing the server_t structure definition.

BUGS

The C_CONF option is not yet implemented.

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

COEDCEis_authorized(rpc)

NAME

COEDCEis_authorized - Makes an authorization decision

SYNOPSIS

```
#include <dcecoe.h>

aud_esl_cond_t COEDCEis_authorized(
        clientid_p_t        clientp,
        unsigned32          auditevent,
        object_p_t          objp,
        dce_and_rec_t *auditrecp,
        error_status_t *status);
```

PARAMETERS

clientp

A pointer to a structure identifying the client whose identity is used for the authorization. This structure is allocated and initialized in the server's manager function. For example:

```
COEDCEclientid_t        clientid;
clientid.identity = ID_HANDLE;
clientid.handle = h;
```

auditevent

If non-zero, this event is used to create an audit event. This event is taken from the message header file produced by sams. For example, if the Segment name was ``CALC", and the CALC.sams file used `cal' as the component name, including ``dcecalmsg.h" would contain the definitions of each audit event.

objp

An object structure describing the object to be located, the permissions required, the type of object, etc. This structure is defined in the application's manager logic. For example:

```
COEDCEobject_t object;
memset(&object, 0, sizeof(object)); /* this is the ACL name in the database
        */ object.name = ``calculator";
/* this is the ACL permission value to test */ object.permname = ``a";
/* this is type of ACL object */ object.obj_type =
        sec_acl_type_default_object;
```

auditrecp

If non-null this audit record structure will be initialized with a start audit call. The caller is expected to commit the audit record. If NULL, the COEDCEis_authorized() function will open and commit the audit event.

status

A pointer to a variable to hold a DCE return status in case a failure condition occurs.

DESCRIPTION

The COEDCEis_authorized() routine makes a decision as to whether or not the client can perform the requested function.

RETURN VALUES

One of the following audit conditions is returned:

aud_c_esl_cond_success
aud_c_esl_cond_failure
aud_c_esl_cond_denial

STATUS CODES

bad_configuration - invalid or incomplete server configuration record

bad_flags - invalid or conflicting flags

SEE ALSO

<dcecoe.h> - for definitions of the COEDCEobject and COEDCEclientid structure.
dce_aud_commit() - the DCE man page for committing an audit record.

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

COEDCElocate_server(3rpc)

NAME

COEDCElocate_server - Locates a server

SYNOPSIS

```
#include <dcecoe.h>

unsigned32 COEDCElocate_server(
        unsigned32          flags,
        unsigned32          service,
        unsigned32          object,
        unsigned32         *count,
        error_status_t *status);
```

PARAMETERS

flags

A set of combinable option flags

service

The index of the service description in the client's configuration record.

object

The index of the object in the service description in the client's configuration record.

count

A pointer used to indicate the number of bindings to return to the client.

status

A pointer to a variable to hold any returned DCE status.

DESCRIPTION

The COEDCElocate_server() routine locates servers based on the service definition in the clients configuration record and makes available a set of binding handles for use in communicating with appropriate servers or for interrogating using COEDCEinquire_server().

FLAGS

C_NOOBJ

Do not return objects in the bindings obtained

RETURN VALUES

bad_parameter - request arguments are malformed

bad_configuration - local DCED configuration record is missing or unusable

bad_flags - invalid or conflicting flags
dce_error - a DCE error is responsible for failure, see dce_error_inq_text(3)

The returned information is a rpc_binding_vector_t obtained using COEDCEgetvector()

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

COEDCEsignal_server(3rpc)

NAME

COEDCEsignal_server - Signal a server to enter listen loop

SYNOPSIS

```
#include <dcecoe.h>

void COEDCEsignal_server();
```

PARAMETERS

None

DESCRIPTION

The COEDCEsignal_server() routine causes a server started with S_WAIT flag to continue, thereby entering its listen loop. This is used when a server should become DCE ready without beginning processing prior to receipt of a special signal.

SEE ALSO

COEDCEinitialize_server()

manual page source format generated by RosettaMan v2.5a6,available via anonymous ftp from ftp.cs.berkeley.edu:/ucb/people/phelps/tcltk/rman.tar.Z

APPENDIX B - SAMPLE APPLICATION

calc.idl

```
/* Sample Application client/server interfaces */

[
uuid(0073a028-fbdb-1e53-908e-08002b13ca26),
version(1.0)
]

interface calculator
{
    import   "dce/database.idl";

    const long calc_s_ok            = 0;
    const long calc_div_by_zero     = 100;

    long
    add (
   [in] long a,
   [in] long b,
   [out]   error_status_t *st
    );

    long
    subtract (
   [in] long a,
   [in] long b,
   [out]   error_status_t *st
    );
}
```

calc.acf

```
/* Sample Application */

[ explicit_handle ]
interface calculator
{
   add([comm_status] st);
   subtract([comm_status] st);
}
```

calc.sams

```
# Sample Application messages for audit events and application
serviceability
#

# Part I
# This part defines the lowest-level table, the one that
contains all the
#  messages (defined in the third part) in a straight array.
component       cal
table           cal__table
technology      dce


# Part II
# This part defines the sub-component table, each element of
which
#  contains the base address of one of the sub-component message
#  tables.
serviceability  table cal_svc_table handle cal_svc_handle
start
    sub-component cal_s_manager "manager"  cal_i_svc_manager
    sub-component cal_s_server  "server"   cal_i_svc_server
end

#
# Part IIa
# This part contains event codes for auditing
#
start
code  add_event
text  "add operation"
action ""
explanation ""
end

start
code  subtract_event
text  "subtract operation"
action ""
explanation ""
end

# Part III
# This part defines the serviceability messages.
#

start
```

```
code            cal_sad_ending
sub-component   cal_s_server
attributes      "svc_c_sev_error"
text            "server initialize failed"
action    ""
explanation  ""
end

#
# Part IIIa
# Messages for serviceability table
#
#  Note that there has to be one of these for each of
#   the sub-components declared in the second part of
#   the file (above)...

start   !intable undocumented
code    cal_i_svc_manager
text    "Manager"
end

start   !intable undocumented
code    cal_i_svc_server
text    "Server"
end
```

CALCclient.c

```
/* Sample client application */

#include <dcecoe/dcecoe.h>
#include "calc.h"

#define SEGMENTSERVICE "CALC"

rpc_binding_handle_t simple(void);
rpc_binding_handle_t complex(void);

main(int argc, char **argv)
{
    unsigned32           err;     /* COE error */
    error_status_t          dceerr;     /* DCE error */
    rpc_binding_handle_t         handle;

    /* interface client logic */
    idl_long_int                    a, b, c;
    char                            operand;
    error_status_t       st;
    int                             rc;

    err = COEDCEinitialize_client(SEGMENTSERVICE, 0, &dceerr);
    if (CHECK(err, "initialize_client", dceerr))
   exit(1);
#if 1
    handle = simple();
#else
    handle = complex();
#endif

    if (handle == NULL) {
   printf("server not installed correctly\n");
        exit(1);
    }

    err = COEDCEstart_server(C_PING|C_START|C_SECURE, 0, handle,
&dceerr);
    if (CHECK(err, "start_server", dceerr))
        exit(1);

    /* user interaction */
    while (true) {
   fprintf(stdout, "Operation: (op val1 val2) ");
   fflush(stdout); fflush(stdin);
```

```
   rc = fscanf(stdin, "%c %ld %ld", &operand, &a, &b);
   if (operand == 'q') break;
   switch (operand) {
   case '+':
       c = add(handle,a,b,&st);
       break;
   case '-':
       c = subtract(handle,a,b,&st);
       break;
   default:
       fprintf(stderr, "Invalid operand\n"); continue;
   }
   if (st == calc_s_ok)
            fprintf(stdout, "%ld %c %ld = %ld\n", a, operand, b,
c);
   else
       CHECK_STATUS(st, "operation failed", CONTINUE);
   (void *)fgetc(stdin);
    }

    COEDCEfinalize_client(0, &st);
}

rpc_binding_handle_t
simple(void)
{
    unsigned32     count = 2;
    unsigned32     err;
    error_status_t dceerr;

    err = COEDCElocate_server(0, 0, 0, &count, &dceerr);
    if (CHECK(err, "locate_server", dceerr) || count < 1)
   return NULL;
    else
   return (COEDCEgetvector())->binding_h[0];
}

rpc_binding_handle_t
complex(void)
{
    unsigned32          count = 100;
    unsigned32          one = 1;
    rpc_binding_handle_thandle = NULL;
    server_t       *servers;
    int         i;
    unsigned32          err;
    error_status_t      dceerr;

    err = COEDCElocate_server(C_NOOBJ, 0, 0, &count, &dceerr);
    if (CHECK(err, "locate_server", dceerr) || count < 1)
```

```
   return NULL;

    for (i=0; i<count; i++) {
        err = COEDCEinquire_server(C_EXEC, 0, &one, &servers,
        (COEDCEgetvector())->binding_h[i], &dceerr);
        if (CHECK(err, "inquire_server", dceerr) || count < 1)
       continue;

   /* pick one based on some criteria */

   handle = (COEDCEgetvector())->binding_h[i];
        err = COEDCEfree_servers(servers, one, &dceerr);
        CHECK(err, "free_servers", dceerr);

   /* we found one we liked */
   if (handle)
       break;
    }

    return handle;
}

CHECK(unsigned32 err, char *msg, error_status_t dceerr)
{
if (err == 0) return err;
    if (err == dce_error)
        dce_printf(dceerr);
    else
   dce_printf(err);
    return err;
}
```

CALCserver.c

```
/* Sample server initialization code */

#include <dcecoe/dcecoe.h> /* for use with COEDCE APIs */

#include "calc.h"

#define SEGMENTSERVICE "CALC"

main(int argc, char **argv)
{
    error_status_t st;

    COEDCEinitialize_server(SEGMENTSERVICE,
           S_LOGIN|S_REFRESH|S_KEYMGMT|S_ACL|S_AUDIT|
           S_CDSEXPORT|S_LISTEN|S_CLEANUP|S_MGMTAUTH,
           NULL, &st);

    exit (st != rpc_s_ok);
}
```

CALCmanager.c

```
/* Sample Manager code - server */

#include <dcecoe/dcecoe.h>

#include "calc.h"    /* build by IDL */
#include "dcecalmsg.h"    /* built by SAMS - audit codes */

idl_long_int
add (rpc_binding_handle_t bh,
    idl_long_int          a,
    idl_long_int  b,
    unsigned32           *st)
{
     COEDCEclientid_tclient;
     COEDCEobject_t     object;

     /* this is how we identify the client */
     client.identity = ID_HANDLE;
     client.id.handle = bh;
```

```
     /* this represents the object to look up and the requisite
perms. */
     memset(&object, 0, sizeof(object));
     object.name = "calculator";
     object.permname = "a";     /* add */

     if (COEDCEis_authorized(&client, add_event, &object, NULL,
st) ==
     aud_c_esl_cond_success)
      return(a+b);

     /* st has status code */
     return -1;
}

idl_long_int
subtract (
    rpc_binding_handle_t bh,
    idl_long_int          a,
    idl_long_int  b,
    unsigned32           *st)
{
     COEDCEclientid_tclient;
     COEDCEobject_t     object;

     /* this is how we identify the client */
     client.identity = ID_HANDLE;
     client.id.handle = bh;

 /* this represents the object to look up and the requisite
perms. */
     memset(&object, 0, sizeof(object));
     object.name = "calculator";
     object.permname = "s";     /* subtract */

     if (COEDCEis_authorized(&client, add_event, &object, NULL,
st) ==
     aud_c_esl_cond_success)
      return(a-b);

    return(-1);
}
```

APPENDIX C - Acronyms

ACL	Access Control List
API	Application Programming Interface
CDS	Cell Directory Service
CFS	Center for Standards
COE	Common Operating Environment
COTS	Commercial off-the-shelf
C^3I	Command, Control, Communications and Intelligence
C4I	Command, Control, Communications, Computers, and Intelligence
DCE	Distributed Computing Environment
dced	DCE daemon
dcecp	DCE Control Program
DFS	Distributed File System
DII	Defense Information Infrastructure
DISA	Defense Information Systems Agency
DNS	Domain Name Service
DTS	Distributed Time Service
dtsd	Distributed Time Service Daemon
ERA	Extended Registry Attribute
GCCS	Global Command and Control System
GCSS	Global Combat Support System
GDS	Global Directory Service
GPS	Global Positioning System
LAN	Local Area Network
NFS	Network File System
NTP	Network Time Protocol
OSF	Open Software Foundation
RPC	Remote Procedure Call
WAN	Wide Area Network